Graphics QuickScreen

Version 1.10

This manual was written by Phil Cramer and Jonathan Waldman with portions excerpted from Don Malin's specifications, The Graphics WorkShop, and QuickPak Professional. This manual was designed and typeset by Jacki W. Pagliaro.

QuickBASIC is a trademark of Microsoft Corp.

CRESCENT SOFTWARE, INC.
11 BAILEY AVENUE
RIDGEFIELD, CT 06877
(203) 438-5300

April, 1994

TOC

Introduction

Installation Instructions

QuickStart

The Screen Designer

Creating Screens

Creating Data Entry Forms

Graphics QuickScreen Routines

Developing In The Basic Environment

Performing Data Entry

Creating Standalone Programs

Graphics Quickscreen Utilities

Product Compatibility

Trouble Shooting

Appendix A

Glossary

Tutorial

1

INTRODUCTION

Thanks!

Thank you for purchasing Graphics QuickScreen from Crescent Software!

We have put every effort into making this the finest and most powerful Graphic Screen building product available. We sincerely hope that you love it. If you have a comment, a complaint, or perhaps a suggestion for another product you would like to see, please let us know. We want to be your favorite software company.

Registration & Upgrades

Please take a few moments to fill out the enclosed registration card. Doing this entitles you to free technical support by phone, as well as insuring that you are notified of possible upgrades and new products. Many upgrades are offered at little or no cost, but we cannot tell you about them unless we know who you are!

Also, please mark the product serial number on your disk labels. License agreements and registration forms have an irritating way of becoming lost. Writing the serial number on the diskette will keep it handy.

You may also want to note the version number in a convenient location, since it is stored directly on the distribution disk in the volume label. If you ever have occasion to call us for assistance, we will probably need to know which version you are using. To determine the version number for any Crescent Software product simply display a directory of the original disk. The first thing that appears is similar to:

```
Volume in drive A is GQS 1.10
```

We are constantly improving all of our products, so you may want to call periodically to ask for the current version number. Major upgrades are always announced, however minor fixes or additions generally are not. If you are having any problems at all, even if you are sure it is not with our software, please call us. As a registered user of one of our products, we provide support for all versions of QuickBASIC and BASIC PDS and can often provide better assistance than Microsoft.

Acknowledgments

Graphics QuickScreen was written by Phil Cramer using Microsoft BASIC PDS 7.1 and library routines from Crescent's QuickPak Professional and Graphics Workshop libraries.

I would like to thank Don Malin for his assistance and advice in developing Graphics QuickScreen. Don wrote the original QuickScreen text mode screen designer that inspired this program. In addition, several of Don's QuickScreen modules were adapted for this graphics version, including the main EditFormG subroutine.

I would also like to thank Brian Giedt for creating many of the graphic assembler routines that make Graphics QuickScreen possible. Several of these routines are derived from Crescent's Graphics Workshop library. Graphics Workshop is highly recommended to anyone interested in further enhancing their graphic applications beyond what BASIC and Graphics QuickScreen alone can accomplish.

- Phil Cramer

Graphics QuickScreen Overview

Graphics QuickScreen is both an EGA/VGA graphic screen design tool and data entry forms library. The screen design component lets you create your own graphic screens—called *display-only screens*—using a sophisticated paint program. Screens are saved in the popular .PCX format and can easily be transported to and from other paint programs. Scanned images saved in .PCX format can also be incorporated into your screens. These screens can be displayed from your BASIC programs at any time.

With Graphics QuickScreen, you can also create screens to be used as data entry forms which we refer to as *data entry screens* or *forms*. This powerful feature lets you to quickly design screens which gather information on a field-by-field basis from a user. Of course, your own BASIC program can control the form and read the values it contains.

Forms are extremely flexible, and data from them can be saved to disk either as simple random access files, or using data base management utilities such as the BASIC PDS ISAM System. Graphics QuickScreen is also fully compatible with AJS Publishing's db/LIB product and Novell's Btrieve library.

Users of QuickScreen

If you are already using Crescent's QuickScreen, you will find the conversion to Graphics QuickScreen to be relatively simple. The screen painting portion of the screen designer is of course different from QuickScreen's text mode counterpart, but the field definition process is virtually identical. In addition, the subroutines that handle your forms use the same or similar calling syntax. Many of the subroutines also have similar names to their QuickScreen counterparts but with the letter G (for Graphics) appended to distinguish between them.

You should also be aware of the QS2GQS.EXE conversion utility. This utility converts QuickScreen text mode screens to graphics modes suitable for use with Graphics QuickScreen. See the section *Graphics QuickScreen Utilities* for more details.

Graphics Mode Screen Designer

To make the task of designing screens as effortless as possible, Graphics QuickScreen's interactive editor is mouse-driven and offers a variety of features:

- Unique pop-up drawing palette.
- Box, radius box, circle, ellipse, arc, polygon and line drawing.
- Multiple line types including user defined.
- User defined grid snap settings.
- Tile palette containing 127 additional dithered colors and 28 tiled patterns.
- Moveable status box indicating the current drawing color, drawing cursor coordinates (relative or absolute), and grid snap status (on/off).
- Block operations such as Copy, Move, and Paste. Any block can be easily centered horizontally or vertically.
- Any block of text may be entered using standard text fonts.
- Scaleable fonts for captions and titles can be displayed at any angle.
- Zoom editor to easily edit individual pixels.
- Palette editor to let you easily assign any of the EGA's 64 colors or the VGA's 256,000 colors to the available 16 color palette.

Data entry screens are created with the help of 23 pre-defined *field types*, such as a zip code and dollar value. Additionally, fields can be further customized:

- Fields can be protected from being changed; they can also be indexed and formatted in any way.
- Fields can support range checks and field calculations based on supplied formulas.
- Unique help messages can be associated with each field in a data entry screen.
- A special data-entry test mode lets you try out the current form during editing. Forms can be generated in two formats.

BASIC Modules

Graphics QuickScreen's BASIC modules allow you to manage both display-only and data-entry screens.

Displaying Screens

To display screens from BASIC you may use a variety of methods:

- Screens may be displayed directly from disk to video.
- EGA Screens can be displayed from video memory using impressive screen wipe effects.
- Partial screen images saved in bitmapped format can be loaded and displayed from memory.

Data Entry

Managing data entry screens from BASIC is one of the more appealing features of Graphics QuickScreen. The supplied BASIC modules let you do the following:

- Control data entry screens automatically based on a *form definition*.
- Handle data entry and movement between fields automatically.
- Perform range checks and field calculations for applicable fields.
- Generate custom help messages for each prompt.
- Support multiple-page forms.
- Polling lets you take special actions based on the user's activity without having to modify Graphics QuickScreen's data entry routines.
- Enable programmers to preset and modify field values, as well as change the cursor position within a form, even at run time.
- Support a mouse without additional programming.

Additional Utilities

Graphics QuickScreen is shipped with two additional utilities you are sure to find useful. The first is a TSR called PCXCAP, a utility which is used

to capture any BASIC- supported graphics screen. Screens captured from EGA or VGA 16-color high resolution screen modes can be later loaded and displayed in the Graphics QuickScreen editor to make further enhancements. Captured screens can be displayed from your BASIC programs using the supplied BASIC modules.

The second utility is QS2GQS.EXE – This program is for owners of QuickScreen and it converts existing .SCR and .QSL text mode screens to graphics modes suitable for use with Graphics QuickScreen. If the screen also contains field definitions, the form definition file (.FRM) is converted as well.

Graphics QuickScreen supports db/LIB, a third-party add-on from AJS Publishing. This library provides routines to read and write dBASE-compatible data files, and, when combined with Graphics QuickScreen's forms, lets you create a powerful graphical database system.

Compatibility

System

Graphics QuickScreen requires an EGA or VGA color monitor and will run on IBM XT, AT, PS/1- and PS/2-class machines and compatibles that contain at least 256k of video memory. At least one megabyte of expanded memory is recommended but not required. DOS 2.0 or above is needed as well as a Microsoft compatible mouse.

Compiler Versions

Graphics QuickScreen is available for users of Microsoft compiled BASIC only: QuickBASIC version 4.x; BASCOM, version 6.x; and the BASIC Professional Development System, version 7.x. If you own an earlier version of BASIC we suggest that you contact Microsoft for an upgrade. We will be happy to assist you in making a decision to upgrade.

Using This Manual

Intended Audience

This manual is designed for users familiar with QuickBASIC and with the concepts of using libraries and compiling to create stand-alone programs. We have not attempted to unnecessarily duplicate information which is QuickBASIC-related and appears in the QuickBASIC documentation, but do explain necessary steps for using this product effectively.

Notational Conventions

We have used some variations in type style mainly so that the manual is clear and more enjoyable to read. The purpose for most type styles is clear (i.e., for topic headings, computer text, and so forth.), however there are some uses which may require further explanation:

- Examples of computer program code are printed in a fixed-spaced font. For instance, consider the DO loop below:

```
'pause for a key press
DO
LOOP UNTIL LEN(INKEY$)
.
.
.
```

Notice also the use of vertical ellipses to convey that more program instructions may follow and the use of BASIC's single-quote REM symbol (') to present comments.

- Examples taken from screen displays are printed as graphic images.

- Pulldown commands are printed in boldface for clarity using this syntax:

  (**Menu name**) pulldown menu command

For example, "**(File) New Screen...**" refers to the **File** menu and the **New Screen...** pulldown command within that menu. When a menu is discussed alone, the menu name, such as **File**, is in boldface.

- DOS directories, file names, acronyms, and BASIC commands are printed in uppercase letters. For example:

```
"The PCXCAP.EXE program is a TSR."
```

- Instances when the computer input or output may vary (depending on your hardware, software, etc.) are shown in italics. For example, the QuickBASIC Quick Library support module will have a slightly different name depending on its version number. We therefore would refer to such a file as in the example below:

```
LINK PROGNAME.OBJ,,BQLB45 /Q
```

Notice that not only is "45" italicized, but also the program name, which is specified by the user, is italicized. Italics in the main text often represent terms which are defined in the glossary.

- In some examples there may be optional features in the syntax. These features will be shown in square brackets. For example, the LET statement is optional in QuickBASIC:

```
[LET] A = 10
```

- Keys on the keyboard are represented as the key name in bold face. Key names are taken from the standard IBM extended keyboard. Certain keys are mentioned in terms of general function. For example, the direction keys typically include the up-, down-, left-, and right-arrow keys, and sometimes the **PgDn** and **PgUp** keys as well. When we need to refer to all of these keys as a group, we will refer to them as direction keys.

Technical Support

If you require technical support for Graphics QuickScreen, you will need your serial number before calling us at (203) 438-5300, between 9:00 a.m. and 5:00 p.m. EST, Monday through Friday. Please gather as much detail as possible about the problem before you call. Be prepared to provide the Graphics QuickScreen version number as well as the BASIC version number. We can assist you best when you are able to describe the precise nature of any difficulties.

Installation

2

INSTALLATION INSTRUCTIONS

Installation

Graphics QuickScreen is distributed using the popular and efficient .ZIP compression format. To help simplify the process of extracting specific files from the archive, we've created a front-end installation program named INSTALL. Upon starting, this program shows the number of bytes the extracted files will occupy, and even allows you to select those files you wish to extract.

To begin installation, place the Graphics QuickScreen distribution diskette in a disk drive. Then, log to that drive and type INSTALL (this example assumes the floppy is in A:):

```
C:\ A:
A:\ INSTALL
```

If you start INSTALL from a drive and/or directory different from the one containing the INSTALL.EXE program, the current drive and directory is used as the installation destination.

When the program starts, it displays its main screen (see Figure 1), and also the name of the product .ZIP files. The second line of the screen displays the available function-key commands. Below this is a field where the installation destination drive and path may be specified. To the right of this is a display field where the amount of free disk space is displayed for the specified target drive. The bottom-left portion of the screen contains a bar menu where the available .ZIP files are displayed along with the disk space required for installation. The bottom line of the menu displays the total disk space needed to install all selected files in the menu. The bottom line of the screen displays comments about the currently-highlighted file.

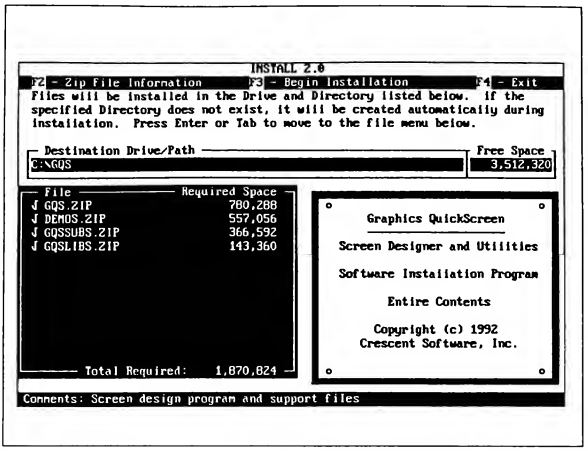


Figure 1: The Installation Start-Up Screen

Several function keys are operable from INSTALL. They are summarized in Table 1.

The Destination Drive/Path field contains a default drive and path where the selected files will be installed. We suggest that you use \GQS as the destination directory. However, you can choose any valid DOS path name. If you specify a path which does not exist, it will be created during installation. If you change the drive letter, the amount of free space on that drive is displayed to the right after moving from this field.

Key	Function	
F2	Info On Zip File	Displays the contents (i.e., file names and sizes) of the currently highlighted file. After viewing or selecting individual files, you can press **Esc** to go back to viewing the actual .ZIP files.
F3	Begin Install	Once you are satisfied with the selected files, you can start the installation process.
F4	Exit	Quits the installation program and returns to DOS.

Table 1: Installation Function Keys

The **Tab** and **Shift-Tab** keys move the cursor between the Destination Drive/Path field and the File Display box. Within the File display, the **Space Bar** or **Enter** key will toggle a check mark (√) on or off. You can check entire .ZIP files or you can press **F2** to check individual files within ZIP files. If you are checking individual files, simply press **Esc** to go back to the .ZIP file list. When you are finished, all checked files are installed when **F3** is pressed.

The number of bytes displayed to the right of each file name is the space required on the destination drive to install that file. This number is the uncompressed size of the file—rounded up to the nearest cluster.

After the Destination Drive/Path has been specified and files have been selected, you can press **F3** to begin installation. If the total expanded size of all selected files exceeds the available disk space, you will be asked whether to continue. You may answer "Y" for Yes if you are sure there is enough space on the target drive. This would be the case if you are installing a newer version of Graphics QuickScreen in the same directory as the one that already exists.

If you are installing to an existing directory you will be asked if you wish to be prompted before existing files are overwritten. We suggest answering Yes if you are not sure about overwriting certain files.

After responding to the prompts mentioned above, the screen is cleared and installation continues. As files are installed, messages from the PKUNZIP.EXE decompression utility are displayed. If another diskette is required, you will be prompted to change the disk in the source drive. After doing this, you can select new files and proceed as before.

Once all selected files have been installed, the program displays a message indicating a successful installation.

Setting The DOS Path

In order to make QuickBASIC and its support files available from any directory, you can set the PATH environment variable from your AUTOEXEC.BAT file. Setting the PATH merely lets you list the directories where your compiler and other executable programs are located. If you are working on a system where several drives are available, you will want to specify the drive letter as well in your PATH statement.

If you install Graphics QuickScreen on C:\GQS, but your version of QuickBASIC is in D:\QB, the DOS PATH variable should be set as follows:

```
SET PATH=D:\QB
```

If your system uses multiple drive letters for several disk partitions, we suggest including the drive letter of each directory in your PATH statement. Doing this ensures that the directory can be properly located.

Some users will need to specify several paths so that QB and BC are properly found. In the Microsoft BASIC Professional Development System, the QBX executable and BC are in separate subdirectories by default: \BC7\BIN and \BC7\BINB, respectively. In this case, you would need to specify both paths:

```
SET PATH=D:\BC7\BIN;D:\BC7\BINB
```

Notice that multiple paths are separated by semicolons. This way, many drive/directory combinations can be searched:

```
SET PATH=D:\QB;C:\DOS;C:\WINDOWS;E:\GAMES
```

The sequence in your PATH statement is significant, because it indicates the order in which the paths are to be searched. And of course, the more entries you have, the longer it may take DOS to complete its search.

To see the current PATH setting, simply type PATH at the DOS prompt.

The Readme File

After installing Graphic QuickScreen, you may want to check for the presence of a README file. Helpful information, as well as additions or changes to this manual, appear in such a file.

After logging onto your Graphics QuickScreen directory, simply enter the following DOS command to view it. (**Ctrl-S** pauses the output until a key is pressed):

```
TYPE README
```

Major Files Of Graphics QuickScreen

The following files are on your distribution diskette; similar file types are grouped together for clarity:

File Name	Description
GQS.EXE	Graphics QuickScreen executable
PCXCAP.EXE	TSR screen-capture program
DBLIB_G.BAS	db/LIB® support module
DEMODBLG.BAS	Demo of db/LIB support routines
DEMOANYG.BAS	Demo which loads a screen and form
DEMOCUSG.BAS	Demo of random-access and form-editing techniques
DEMOINVG.BAS	Demo containing fields with multiple-choice array, calculated fields, and multi-line text
DEMOPAGG.BAS	Demo illustrating the use of a two-page form
EVALUATE.BAS	Double-precision equation handler
EDITFORM.BAS	Form data entry handler
FRMFILE.BAS	Loads information from form (.FRM) files
GQSCALC.BAS	Used for calculated fields in a form
GDISPLAY.BAS	.PCX screen display module
NOCALCG.BAS	Used to exclude support for calculated fields
NOMULTG.BAS	Used to exclude support for multiple-choice fields
NONOTESG.BAS	Used to exclude support for notes fields
NOSCROLLB.BAS	Used to exclude support for scroll bars
NOSCROLL.BAS	Used to exclude support for scrolling text fields
GFORMS.LIB	Graphics QuickScreen assembler library file for QuickBASIC 4.x or BASIC 6.0
GFORMS7.LIB	Graphics QuickScreen assembler library file for BASIC PDS 7.x

GFORMS.QLB	Graphics QuickScreen quick library file for QuickBASIC 4.x or BASIC 6.0
GFORMS7.QLB	Graphics QuickScreen quick library file for BASIC PDS 7.x
TILEPAL.GM4	Tile Palette bitmap
TPAL.TIL	Random file containing tile definitions for the Tile Palette
*.BI	BASIC Include files
*.FRM	Graphics QuickScreen Form files
*.MAK	Graphics QuickScreen Make files

File names that start with the letters NO are used to exclude support for certain features, and serve a similar function as the stub files that come with some versions of BASIC. Stub files replace selected modules with others having the same name but have reduced functionality. This lets you reduce the size of your programs when certain features (such as calculated fields) are not needed.

Copying And Backing Up

Before you start to use the program you should first make a copy of the original diskette and then work with the copy. We know we don't have to tell you this, but reminding you may prevent a very frustrating situation should your distribution diskette become damaged.

Quick Start

3

Quick Start

QUICK START

If you are familiar with QuickBASIC and add-on libraries, you can get started quickly by running the Graphics QuickScreen demonstration programs. These programs are liberally commented so you can easily see how they work and how the Graphics QuickScreen routines are set up and called.

Running The Demos

Graphics QuickScreen includes several demonstration programs. We encourage you to both experiment with these programs and copy statements from the demos into your own programs.

Starting with the simplest among them, the programs are:

- DEMOANYG.BAS

This is a basic example of how to load a form definition file, display a screen, and allow the user to perform data entry. We've called it DEMOANYG since the program can work with any standalone screen (.PCX) and form (.FRM) file.

- DEMOALLG.BAS

This example is particularly useful because it displays a form which contains each Graphics QuickScreen field type.

- DEMOCUSG.BAS

This example provides a customer information form, and shows how to store and retrieve information using random access file techniques. This program also provides a technique for clearing all fields so that a fresh form can be presented to a user.

- DEMOINVG.BAS

This demonstration shows an invoice form and the special features which make Graphics QuickScreen so powerful, such as the use of multiple-choice, calculated, and note fields. Examples of advanced polling are demonstrated which report the user's activity on the form and make certain fields change their characteristics based on user-defined options.

- DEMOPAGG.BAS

Demonstrates using EditFormG to allow data entry on a multi-page form. This program also demonstrates loading screen and form files from a custom .GSL library.

- DEMODBLG.BAS

This demonstration of an employee information form requires db/LIB, a product by AJS Publishing, which allows BASIC programmers to read and write dBASE-compatible files.

To run a demonstration program, you can follow these steps:

1. Change to your Graphics QuickScreen directory:

```
CD \GQS
```

2. Start BASIC, making sure to specify the appropriate quick library, such as GFORMS.QLB:

```
QB /L GFORMS
```

For BASIC 7.x, use the following:

```
QBX /L GFORMS7
```

These Quick Library files contain the various assembly language sub-routines used by the Graphics QuickScreen modules. Therefore, they are needed to run any of the supplied programs.

If you need to use additional routines from other libraries to your program, you can use the MAKEQLB utility that comes with Graphics QuickScreen to create them.

3. Select the **(File) Open** menu command, then choose the demonstration you wish to run from the dialog box which appears.

4. Once the program has been loaded, you can press **Shift-F5** to run it.

Screen Designer

4


Screen
Designer

THE SCREEN DESIGNER

Graphics QuickScreen's editor, referred to as a *Screen Designer*, is provided as a compiled executable program called GQS.EXE. It is possible to start this program immediately from the DOS prompt simply by typing "GQS".

Once the program begins you should be able to use Graphics QuickScreen's intuitive interface right away. If you have a mouse, it will be recognized and used automatically. Also, a help screen showing the action of various keys is available by pressing **F1**.

Graphics QuickScreen's user-interface is based on a convenient pop-up Drawing Palette, a comprehensive pulldown menu system and dialog boxes. The Drawing Palette allows instant access to the color palette and to the most commonly used drawing tools. The menu system organizes the major command categories as menu titles and pulldown commands. Dialog boxes query the user for additional information before certain commands are performed. These features are described in detail in the sections that follow.

The Drawing Palette

The Drawing Palette is used to select the drawing color and other commonly used drawing and editing tools that you will use to create your screens. It has been designed as a pop-up to let you to see the entire screen when it is not in use.

During drawing and editing, the right mouse button and the **Esc** key both work as toggles to turn the Drawing Palette on and off. The Drawing Palette always pops-up beneath the graphic cursor and it remains active until a drawing tool has been selected or the right mouse button is clicked. Once a tool has been selected, you can continue to draw or edit using the left mouse button. To cancel the current tool, click the right mouse button to return to the Drawing Palette.

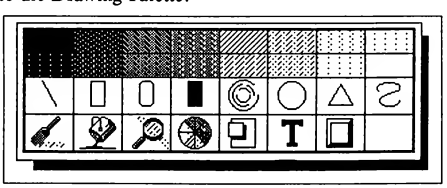


Figure 2: The Drawing Palette

The Drawing Palette may be accessed at any time through a series of no more than three right mouse clicks.

Selecting a Color

To select a drawing color from the Drawing Palette, place the cursor over the desired color and click the left mouse button. The selected color will appear in the lower right corner of the Drawing Palette. The color will also appear in the Status Box if it is active. In addition, you may select a color by pressing its corresponding numeric key. Numeric keys correspond to the standard color assignments as used by BASIC where **0** = black, 1 = blue, 2 = green, 3 = cyan, and so on. To access colors **10** through 15, hold down the **Shift** key as you press the numeric keys **0** through **5**.

Colors may be changed during any of the drawing operations, even in the middle of a paint or sketching procedure by typing its assigned color number. (Numeric color keys are not supported when typing text selected from the **T** icon or from the **(Draw) Print Text** menu because they are used to enter the actual numbers as text.)

General Notes On Drawing

All drawing procedures are handled in a similar fashion. To begin a procedure position the cursor and click the left mouse button. This will produce a "rubberband" line, circle, or box that can be positioned as you like. Once you are satisfied with the placement of the item, complete it by clicking the left mouse button. You may continue to draw using the same tool, or you can exit back to the Drawing Palette by clicking the right mouse button.

In general, the left mouse button is used to initiate or complete an action while the right mouse button is used to cancel it. Once a drawing or editing operation is canceled, one more right mouse click will return you to the Drawing Palette.

The mouse cursor can also be controlled from the keyboard by using the various cursor keys. The **Enter** and **Esc** keys perform the same function as the left and right mouse buttons.

Any edits you make can be undone by pressing **F10**. This will restore the screen to its condition just before the last drawing tool or menu item was selected.

Lines

→ To draw a line:

1. Select the line icon from the Drawing Palette.
2. Place the cursor where you want the line to start and click the left mouse button.
3. Move the cursor to the desired end point and click the left mouse button.

Line drawing will continue from the last endpoint until you click the right mouse button. A single right click lets you start a new line and two clicks returns you to the Drawing Palette.

Polar Mode Line Drawing

Polar Mode allows you to draw lines a specified length and at a specified angle.

→ To draw lines in polar mode:

1. Select the line icon from the Drawing Palette.
2. Select a starting point.
3. Press the **P** key at any time during the line drawing procedure and a dialog box will appear.
4. Enter the desired length (in pixels *) and the angle in degrees separated by a comma and then click "OK". The length should always be a positive value but degrees may be specified as any positive or negative whole number.
5. Click the left mouse button to accept the line and to continue drawing in polar mode. Clicking the right mouse button will undo the line and return you back to conventional line drawing.

End coordinates that would place the line off-screen are not allowed. In this case, a beep warning will sound and Graphics QuickScreen will revert back to normal line drawing.

* On a VGA monitor (640x480) where the aspect ratio is 1:1 (10 vertical pixels appear the same length as 10 horizontal pixels) the length at any angle will be consistent. When using an EGA monitor where the aspect ratio is approximately .73:1, 10 horizontal pixels do not represent the

same length as 10 vertical pixels. In this case the length specified is in horizontal pixels regardless of the angle specified.

Box

→ To draw a box:

1. Select the box icon from the Drawing Palette.
2. Place the cursor at any corner where you wish to start the box and click the left mouse button.
3. Move the cursor until the box is the size desired and click the left mouse button.

You may continue to draw boxes or return to the Drawing Palette by clicking the right mouse button.

Radius Box

The procedure for drawing a radius box is the same as for normal box drawing. The radius for the corners is measured in pixels and is set from the **System** dialog box. If the box dimensions are less than two times the radius of the arc, the radius is automatically reduced to fit the dimensions. A very large radius therefore enables you to draw virtually any size oval without readjusting the radius.

Filled Box

The procedure for drawing a filled box is the same as for box drawing. Also see the section *Painting Fields* for more information.

Arcs

→ To draw an arc:

1. Select the arc icon from the Drawing Palette.
2. Position the cursor at the center of the arc and click the left mouse button.
3. Locate the starting point of the arc with the cursor and click the left mouse button. A circle will appear defining the possible path for the arc and an "X" will appear at the selected starting point for the arc.

4. To define the end point for the arc, place the cursor such that the "rubberband" radius intersects the circle at the desired end point, and then click the left mouse button. This point can be anywhere on the screen, even inside the circle. The circle and the "X" mark will disappear and an arc will be drawn counterclockwise from the starting point to the specified endpoint.

If you prefer to draw arcs clockwise from the starting point, press the **A** key while in the arc drawing mode. The cursor will change color (from white to yellow if on a black background) and a beep will sound to acknowledge your selection. Arcs will then be drawn clockwise from the starting point. Pressing **A** again will toggle back to counterclockwise operation. Elliptical arcs are not directly supported but can be simulated by using the **Draw Polygon** procedure.

Circle/Ellipse

→ To draw a circle or an ellipse:

1. Select the circle icon from the Drawing Palette.

2. Position the cursor at the center of the circle and click the left mouse button.

3. Move the cursor until the circle is the size desired and press the left mouse button. To ensure a true circle, drag the mouse just out side of the circle before releasing the mouse button.

4. To draw an ellipse, hold the mouse button down and *push* or *pull* the circle into an ellipse. The type of ellipse drawn will depend on where the cursor is when you press the button. If pressed at approximately 12 o'clock or 6 o'clock you can draw a horizontal ellipse. If you press at approximately 3 o'clock or 9 o'clock you will draw a vertical ellipse. If you press at approximately 1:30, 4:30, 7:30 or 10:30 then you will be able to draw either a vertical ellipse or a horizontal ellipse, depending on the position of the cursor as you drag it inside the encompassing circle. Release the button to complete the ellipse.

Polygons

The polygon routine will draw a polygon with from 3 to 99 sides at virtually any size and at any angle. You can also specify an aspect ratio to stretch the polygon either vertically or horizontally. You can also limit the number of sides of the polygon that are displayed.

→ To draw a polygon:

Select the triangle icon from the Drawing Palette. A dialog box will appear prompting you for the following information:

Number of sides: The total number of sides in the polygon *.

Starting Angle: The angle where drawing starts. The default angle is 0 and corresponds to the 3 o'clock position. Angles are then measured from this point counterclockwise with 90 degrees being vertical (12 o'clock), 180 being horizontal (9 o'clock), and so forth. The desired angle can be any whole angle, positive or negative.

■ **Example:**

To draw a standard isosceles triangle with the base at true horizontal, specify three sides and a starting angle of 90. A standard square can be drawn by specifying four sides and a starting angle of 45 degrees or you can draw a diamond by leaving the starting angle at 0.

Sides to display: The number of sides to display. This can be any number from 0 to the total number of sides. A value of zero will display all sides as will a value equaling the total number of sides. Any other value less than the total number of sides will display only that portion of the polygon beginning at the starting angle.

X/Y Ratio: The aspect ratio of the polygon. This parameter allows you to stretch the polygon. An aspect greater than 1 stretches the polygon horizontally, while a value less than 1 stretches the polygon vertically.

■ **Example:**

By changing the aspect ratio and the starting angle of a triangle you could easily create different triangle styles as arrow heads. You could also change the aspect ratio, specify a relatively large number of sides and display only a portion of the total number of sides to simulate an elliptical arc. Many other interesting shapes are possible with various combinations of these parameters.

* Numbers greater than 20 or so produce what essentially appears to be a circle and the "rubberbanding" effect in that case is considerably slower than using the circle command.

Sketch

The **Sketch** routine lets you draw freehand images by dragging the mouse or by using the various cursor keys.

1. Select the sketch icon from the Drawing Palette.

→ To sketch lines using a mouse:

2a. Position the cursor where you want the line to begin and hold down the left mouse button. As long as the button is down, a continuous line will be drawn as you drag the mouse. Releasing the button will complete the line.

→ To sketch lines using the keyboard:

2b. Position the cursor where you want the line to begin and press **Enter**. Line drawing is controlled by pressing the various direction keys. The mouse can still be used to control line drawing but it is not necessary to hold down the left mouse button. Pressing **Esc** or clicking the right mouse button will complete the line.

You can continue to sketch lines or click the right mouse button to return to the Drawing Palette.

Paintbrush

The **Paintbrush** routine lets you paint lines of nearly any thickness by dragging the mouse or by using the various cursor keys. The brush size can be set from the System dialog box.

→ To use the Paintbrush:

1. Select the paintbrush icon from the Drawing Palette. A rectangular cursor will appear indicating the size of the paintbrush.

→ To paint using a mouse:

2a. Position the cursor where you want to begin painting. The paintbrush is activated whenever the left mouse button is held down.

→ To paint using the keyboard:

2b. Position the cursor where you want to begin painting and press **Enter**. The paintbrush is controlled by pressing the various direction keys. The mouse can still be used to position the paintbrush but it is not necessary to hold down the mouse button. Pressing **Esc** or clicking the right mouse button will stop the paint flow.

You can continue to paint or click the right mouse button to return to the Drawing Palette.

Flood Fill

The **Flood Fill** routine allows you to quickly and easily paint entire regions of any shape. The region being filled must be *entirely* surrounded by a single color to contain the flood of paint. When you click on an area to be filled, the routine first checks the color beneath the cursor and then looks to the right to find the first occurrence of a different color. It assumes that the second color it finds is the surrounding color and floods the entire region surrounded by that color.

→ To flood fill a region:

1. Select the paint bucket icon from the Drawing Palette. The cursor will change from white to black.

2. Locate the cursor within the surrounding color and click the left mouse button.

You may continue to flood fill regions or return to the Drawing Palette by clicking the right mouse button.

Zoom Editor

The **Zoom Editor** allows you to zoom in on a region of the screen and easily edit individual pixels. You may select any rectangular region of the screen up to approximately 1-½ square inches depending on the monitor size and screen mode. The region will be identified with a surrounding box and the enlarged portion will automatically display near the selected region. The screen will be updated simultaneously to reflect any edits you make.

→ To edit individual pixels:

1. Select the magnifying glass icon from the Drawing Palette. A drawing cursor will appear.

2. Draw a box surrounding the area to be magnified following the box drawing procedure. If the region is within range, it will be magnified and you may begin editing. A beep will sound if the selected region is too big to be enlarged. In that event, simply select a smaller region.

3. Editing is performed by clicking the left mouse button at the desired location in the enlarged image. Colors are selected by using the number keys as explained in the *Selecting a Color* section of this manual. The **Status Box** can be used to display the current drawing color.

You can undo any edits by pressing **F10** while the zoom window is still active.

Recolor

The **Recolor** option lets you change any color in a specified region to a new color.

→ To recolor a region:

1. Select the color wheel icon from the Drawing Palette. A color palette will appear replacing the Drawing Palette and the prompt *"Pick the color to change"* is displayed.

2. Pick the color to change by clicking on it with the left mouse button. The selected color will appear in the lower right window of the color palette and the message will change to *"Now pick the new color"*.

3. Select the new color by clicking on it with the left mouse button. The color palette will disappear and a drawing cursor will appear.

4. Draw a box surrounding the region to be recolored following the box drawing procedure. The area will be recolored as soon as the box is completed.

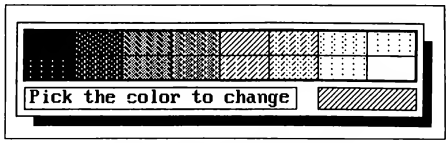


Figure 3: The ReColor Palette

You may continue to recolor portions of the screen or you can return to the color palette to select new colors to change by clicking the right mouse button. One more right mouse click will return you to the Drawing Palette.

Copy/Move

The **Copy/Move** option allows you to copy or move any region of the screen to memory (up to 64k) or to any place on the screen. The image can also be saved to disk using the **Save Paste Buf...**" option from the pull-down menu or later recalled by selecting the **Paste** option from the **Edit** menu.

The **Move** option lets you grab an image and move it anywhere on the screen leaving the background color* in its place. The **Copy** option lets you make multiple copies of an image without deleting the original. The default mode when selected from the Drawing Palette is **Copy**. You can switch the default to Move by changing the setting on the **System** dialog box. **Copy** and **Move** are also available separately from the **Edit** pull-down menu. Regardless of what mode was used to capture a region, the image will remain in the paste buffer even after loading a new screen or until one of the following occur:

1. A new image is captured
2. The **Tile Palette** is selected
3. **"Try data entry in form"** is selected

→ To copy or move a region:

1. Select the copy/move icon from the Drawing Palette. A drawing cursor will appear.
2. Draw a box surrounding the area to be copied or moved using the box drawing procedure. Click the left mouse button to capture the

image. You can center the image horizontally or vertically by pressing the **H** or **V** keys respectively.

3. Locate the image where desired and click the left mouse button. If **Copy** is selected, you can continue to make copies by clicking the left mouse button. If **Move** is selected, a single left click will complete the procedure.

The captured image is moved over the screen using XOR. As the image is moved, you see a combination of the copied image and the underlying background with colors inverted as they are XORed together. When pasted, the image will return to its original colors. (XOR off). Checking "XOR on" on the **System** dialog box under the **Settings** menu will instead use XOR when pasting the image. This is particularly useful when the background of the captured image is black. In this case, only the colored portion of the captured image will be copied; the rest of the image remains transparent. Unwanted color changes in the pasted image can be corrected using the recolor command.

* Since graphic modes do not directly support both foreground and background colors, one must be specified in this case. The background color for the move option is assigned in the **System** dialog box found under the **Settings** menu. The default color is black but can be assigned to any of the sixteen colors.

Print Text 

This option lets you enter text using your computers internal font at standard row and column coordinates. The text is printed without disturbing the underlying screen.

→ To print text:

1. Select the desired text color and click on the **T** icon from the Drawing Palette. A blinking text cursor will appear.

2. Point the mouse cursor at the desired row and column and click the left mouse button.

3. Type in the desired text. Text can be erased without affecting the underlying screen using either the **Spacebar** or **Backspace** key as long as you remain on the same line.

The following table lists the keys that can control the cursor while printing text.

Key	Action
Left arrow	One space to the left
Right arrow	One space to the right
Up arrow	Up one row
Down arrow	Down one row
Tab	8 spaces to the right
Shift + Tab	8 spaces to the left
Ctrl+ Left	20 spaces to the left
Ctrl + Right	20 spaces to the right
Home	Moves the cursor to the first column
End	Moves the cursor to the last column
PgUp	Moves the cursor to the top row of the screen
PgDn	Moves the cursor to the bottom row of the screen

Table 2: Text Cursor Control Keys

Push Button

The Push Button will draw a 3-dimensional push button in the color specified. Creating the 3D effect requires the use of three shades of the same color: one for the highlight, one for the button color, and one for the shaded portion. The default palette provides two shades of gray plus high-intensity white. This is an ideal combination for push buttons and is perhaps why gray buttons are so common. You can of course draw push buttons using any combination of colors from the EGA or VGA color palette.

The default palette offers two intensities of blue, green, cyan, red, violet, and brown, which you can also use to draw push buttons. When a color is selected from the Drawing Palette, the high-intensity shade will be used as the highlight color and the low-intensity shade will be used as the actual button color. The shaded portion of the push button will always be painted dark gray. For the best effect, the dark gray portion of the button should be changed to a darker shade of the button color. Since this darker color does not exist in the standard palette, one of the other colors will have to be modified to replace it. This can be accomplished by using the Palette Editor to select the new color. Use the Recolor option to replace the dark gray portion with the new color.

You can also draw any image or place text on the push button as long as its border is at least 2 pixels from the highlighted or shaded portion of the push button. Please refer to Figure 4.

→ To draw a push button:

1. Select the push button icon from the Drawing Palette.
2. Place the cursor at any corner where you wish to start the push button and then click the left mouse button.
3. Move the cursor until the box is the size desired and then click the left mouse button. The push button will be drawn in the selected color.

You may continue to draw push buttons or return to the Drawing Palette by clicking the right mouse button. Note that at this point the button is just a graphic image and needs to be defined as a field* before it will actually function as a push button.

*We recommended that you draw push buttons with the grid snap on. This greatly simplifies the process of later defining them as fields.

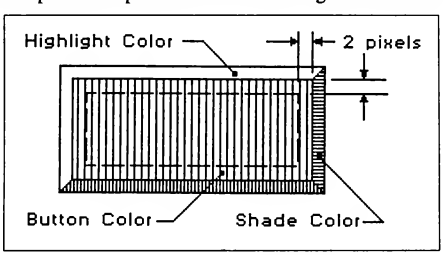


Figure 4: Pushbutton Detail

Several other drawing tools are available, but they can be accessed only from the pull-down menu. See the section *Using the Menu System* for information on using these tools.

Drawing Aids

Controlling The Mouse Cursor

The Graphics QuickScreen drawing program works best when controlled with a mouse. However, all commands as well as control of the graphics cursor can be accessed from the keyboard.

The following table lists the keys that control the mouse cursor during any of the drawing or editing procedures. The X and Y snap space refer to the current spacings set for Grid Snap.

Key	Action
Left/Right arrow	If snap is on, moves cursor one X snap space right or left; if snap is off moves cursor one pixel right or left
Up/Down arrow	If snap is on, moves cursor one Y snap space up or down; if snap is off, moves cursor one pixel up or down
Tab	Moves cursor 80 pixels to the right
Shift + Tab	Moves cursor 80 pixels to the left
Ctrl + Left/Right	Moves cursor 72 pixels to the left or right
Ctrl + Up/Down	Moves cursor 4 rows up or down
Home	Moves the cursor to the left-most pixel, on the current row
End	Moves the cursor to the right-most pixel, on the current row
PgUp	Moves the cursor to the top of the screen in the current column
PgDn	Moves the cursor to the bottom of the screen, in the current column
Enter	Same as left mouse click
Escape	Same as right mouse click

Table 3: Cursor control keys

Grid Snap

The **Grid Snap** option allows you to set the X and Y increments allowed for cursor movement. Increments are specified in terms of pixels and can be set to virtually any number. This can greatly simplify the process of making accurate drawings.

→ To set the grid snap spacing:

1. Select **System** from the **Settings** pull-down menu. A dialog box will appear.

2. Set the desired spacings in the text boxes labeled **XSnap spacing** and **YSnap spacing**.

Grid Snap can be toggled on and off at anytime during drawing or editing by pressing the **S** (Snap) key. If active, the Status Box will indicate the current snap status by showing the labels in upper case if snap is on, or

lower case if it is not. On start-up or after selecting a new screen mode, X and Y Grid Snap spacings are automatically set to correspond with the standard text size for the current screen mode.

Painting Fields

Data entry fields use your computer's internal ROM fonts to display text. The width of these fonts is always eight pixels but their height will vary based on the screen mode you select. For 25, 30, 43, and 60-line modes, the height will be 14, 16, 8, and 8 pixels high respectively.

When designing forms, it is common to paint the data entry fields a unique color so that users can clearly see where the fields begin and end. Graphics QuickScreen makes this a simple process by allowing you to define grid snap spacings that correspond exactly to the standard text spacings. With the proper settings you can use the Filled Box routine to paint regions that correspond exactly to the size required by the data input routines. The color you select for painting these regions will become the field's background color. The field's foreground color is assigned during the field definition process.

The table below lists the grid snap settings to use when painting fields for the different screen modes.

Screen Mode	Number Of Rows	Text Size*	X /Y GridSnap
640x350 EGA	25	8x14	x = 9, y = 15
640x350 EGA	43	8x8	x = 9, y = 9
640x480 VGA	30	8x16	x = 9, y = 17
640x480 VGA	60	8x8	x = 9, y = 9

** Text size in pixels*

Table 4: Grid Snap Settings for Painting Fields

You can set these values in the **System** dialog box, or they can be set automatically by pressing the **F** key while drawing or editing. The **F** (Field Paint) key toggles between the current grid snap settings and the appropriate settings from the table above. As you press the **F** key, you will hear two different beep tones. When you hear the higher pitch, grid snap is set to match text coordinates. The Status Box will display a black rectangle when grid snap settings correspond to the current text size.

Note that when using the **Filled Box** routine with these settings, a special condition applies. The height and width of the filled box will be one pixel less than with any other setting. This is necessary to duplicate the exact

size of the background for the font that will be used for data entry fields. This condition applies only to the settings that correspond to the text size for the currently selected screen mode.

In the following example, it is assumed that we are working in VGA 25 line mode. The drawing on the left shows the effect of the **Filled Box** routine when grid snap settings correspond to the current text size: 9 and 17 (in pixels). Note how the painted box is one pixel less than the actual snap coordinate on the bottom and right sides, and that adjacent boxes do not overlap. The drawing on the right shows the effect of the Filled Box routine when grid snap coordinates are different from the text size. In this case the painted region corresponds exactly to the grid snap settings.

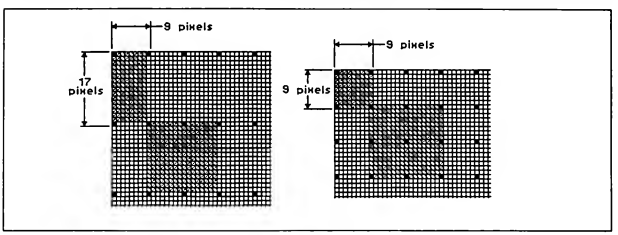


Figure 5: Effect of Grid Snap settings on Filled Box

We recommend that you draw with grid snap on whenever possible. Drawing will be faster and more accurate, since mouse movement needs to be less precise. Cursor movement will also be faster when using the direction keys since the cursor moves in increments that correspond to the current grid snap settings. Defining push buttons, mouse fields, and scroll bars as fields is greatly simplified when defined using the same grid snap settings used to draw them.

The Pulldown Menu System

Figure 6 shows the Graphics QuickScreen display with an active menu system. The menu system is said to be active whenever the menu bar is visible. On the first line of the screen appears the menu bar, and under each menu bar option is a unique pulldown menu. Most major commands in Graphics QuickScreen are available by accessing this menu system, or by using shortcut keys which execute a command with one keystroke when the menu system is not active.

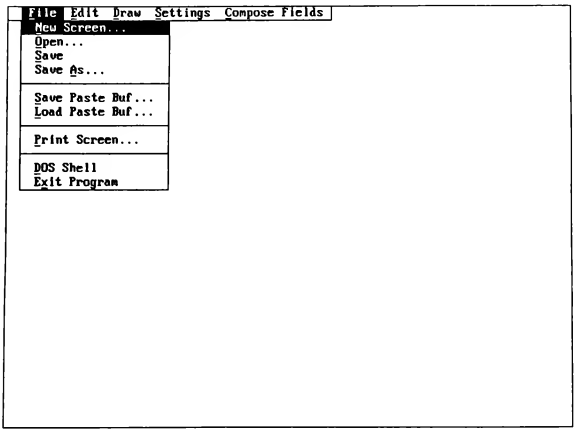


Figure 6: File Pulldown Menu

Menu System Contents

Table 5 below summarizes the pulldown menu features. The menu system reflects a user-interface with which you are most likely already familiar. Very much like BASIC's own menu system, Graphic's QuickScreen menus may be used with the keyboard or mouse. For this reason, these discussions are presented separately below.

Menu Name	Pulldown Menu Command Features
File	This menu includes load and save options in two formats, prints the screen, executes a DOS shell, and exits Graphics QuickScreen.
Edit	This menu contains routines to Copy, Move and Paste images, flip images horizontally or vertically, and it also measures screen coordinates.
Draw	This menu lets you print text using standard ROM fonts, print text using scalable fonts that can printed at any angle, draw smoothed open or closed curves, and draw horizontal or vertical scroll bars.
Settings	This menu selects the drawing cursor, line type, Palette Editor, Status Box, System settings, and Set Paths dialog boxes.
Compose Fields	From this menu you may define, move, copy, rearrange and print data field definitions. You may also generates a BASIC source file to display and edit your forms.

Table 5: Summary of Menu Commands

Keyboard

The keyboard interface to the menu system is very extensive. In general, the direction keys are used to select a menu and a pulldown command. Once a command is chosen, **Enter** is pressed to execute it, or **Esc** is pressed to abandon the choice. Although it is common to rapidly develop a "feel" for the menu system, the summary in Table 5 may assist you in learning even faster. Please notice that some keys have more than one function.

If you press the **Alt** key when Graphics QuickScreen begins, the menu bar depicted in Figure 6 will appear. At this point you may scan across the menu bar by using the **Left** and **Right** direction keys. Once the desired menu (such as **File**) is selected, you may press the **Up** and **Down** keys until the desired command (such as **New Screen...**) is highlighted. To execute the command simply press **Enter**, or press the highlighted letter corresponding to the command of your choice.

Once the screen editor is in use, the menu system is deactivated and the menu bar will no longer be visible at the top of the screen. At this point you may activate it by pressing **Alt**, or by clicking the left mouse button, which will show the last pulldown menu used. Alternatively, you may access a particular pulldown menu by pressing **Alt** plus the first letter of the desired menu name. For example, to access the **File** pulldown menu, press **Alt-F**.

Mouse

When pressed, the left mouse button toggles the menu bar on and off the screen. Once the menu bar is displayed you may move the mouse cursor over a menu item and press the left mouse button. To choose a pulldown menu command, simply move the mouse cursor over the desired option and click the left mouse button.

Some users may find that "dragging" the mouse produces better results: Move the mouse cursor over the desired menu bar title and press and hold down the left mouse button. You may then select different pulldown menu commands simply by moving the mouse cursor along the pulldown menu. If the mouse button is released while the mouse cursor is over a command, then the command selected will be executed. You may also drag the mouse cursor along the menu bar to view other pulldown menus before making a selection.

Key(s)	Action on Menu System
Alt	Displays the menu bar; highlights the menu hot keys
Alt-char	Generates the menu of the menu bar option starting with the character pressed; executes a command having the hot key of the character pressed
char	When the pulldown menu is displayed, pressing a character accesses and executes the pulldown menu command having the same highlighted or underlined letter as the character pressed.
Right/Left	Selects menu options on the menu bar
Up/Down	Selects commands in a pulldown menu
Enter	Generates the pulldown menu of the menu bar option selected; executes the pulldown menu command selected

Table 6: Menu System KeyBoard Interface Summary

If you do not wish to choose a command after you have activated the menu system, simply move the mouse cursor away from the menu system. Then either click or release the left mouse button or press Escape.

General Comments

Note that some pulldown menu commands are black while others are gray. The black commands are active; the gray commands are inactive and will produce no effect if selected. Once a form is loaded into the Graphics QuickScreen editor many of the inactive choices will become active. For example, under the **File** menu, the **Save Paste Buffer...** command will not be active until something is loaded into the paste buffer.

Dialog Boxes

A pulldown menu choice followed by ellipses usually generates a dialog box. Dialog boxes provide an effective way to gather information from the user, and makes it easy to enter information or to select options.

Figure 8 depicts the Open File dialog box and highlights three major dialog box input elements: the text box, list box, and command buttons. The text box accepts a string of characters from the keyboard and allows the entry of a path and file name. The list box presents items in a columnar list. List boxes may hold many items and their contents may be scrolled using the direction keys or scroll bar. The command buttons carry out the designated command when chosen. Note that pressing **Enter** or **Esc** is equivalent to clicking either the OK or Cancel command buttons respectively.

Following is a brief summary of the keyboard and mouse interface used by the Graphics QuickScreen dialog box input elements.

Keyboard

When a dialog box is first presented the cursor will rest on a particular input element. This cursor, or input focus, may be moved to the next input element by pressing **Tab**, or to the previous input element by pressing **Shift-Tab**. The input focus may also be directed to a particular input element by pressing the underlined or highlighted hot key for that element. To direct input focus while in a text or list box use the **Alt-hot key** combination.

Aside from these general directions, there are more specific ways to use each dialog box input element with the keyboard:

- **Text Box**

  The Text Box accepts text which is typed by the user. When you first enter a text field, typing any text character will clear the field and start a new word. If you wish to edit the current contents of the text box without clearing the field, press any of the direction keys before entering any text. Selecting the field with the mouse also allows editing the existing text without first clearing the field. You can also clear the field by pressing **Ctrl-C** or restore the original contents by pressing **Ctrl-R**.

- **Multi-Line Text Box**

  The multi-line text box accepts text which is typed by the user. Text is automatically word-wrapped as it is entered. Standard editing keys such as **Home**, **End**, **PgUp**, **PgDn** and direction keys are also supported. You can delete an entire line of text by pressing **Ctrl-Y**. Deleted text may be inserted by pressing **Shift-Ins**.

- **List Box**

  List box items are selected with the direction keys or by using the scroll bar. When the desired item is selected you may press **Enter** to accept it.

- **Check Box**

  The check box is toggled by pressing the **Spacebar**.

- **Option Button**

  An option button is selected for a specific group by pressing the **Up** and **Down** cursor direction keys.

- **Command Button**

  You may execute the highlighted Command Button at any time by pressing **Enter**. You may also use **Tab** to access a particular command button and press **Enter**. Further, pressing the underlined or highlighted hot key of a command button will also execute it.

Mouse

If you have a mouse, you may access dialog input elements by clicking on the desired element. More detailed instructions are summarized below.

- **Single and Multi-line Text Boxes**

  The mouse is not useful for entering information into a text box. You may, however, direct the input focus to or locate the cursor in a text box by clicking on it.

- **List Box**

  A list box item may be selected by double-clicking on it. Double-clicking refers to pressing the left mouse button twice in rapid succession. If a list box has more information, its contents may be scrolled by clicking the mouse on the scroll bar. You can also select an item by pressing a key that corresponds to the first letter of a menu item.

- **Check Box**

  The check box is toggled by clicking it with the mouse.

- **Option Button**

  An option button is selected by clicking on it with the mouse.

- **Command Button**

  You may execute a command button by clicking it with the mouse. Clicking the right mouse button exits the dialog box as if **Cancel** had been selected.

  All dialog box elements can be accessed by pressing the underlined or highlighted hot key. If you are currently in a text box, multi-line text box or list box, you must press **Alt** in addition to the hot key.

Menu Item Information

This section presents an overview of the menu system commands. Menu bar options are organized into subsections; corresponding pulldown commands are listed next to round bullets. The underlined letter in the commands discussed below represents the hot key for the command. These hot keys appear on-screen as underlined or bright letters depending on the screen mode, and they allow immediate access to an item merely by pressing the highlighted key. Finally, pulldown menu choices followed by ellipses usually present an editing palette or a dialog box for further input.

This section attempts to be exhaustive. Further explanation, however, may be encountered when a particular command is discussed later in the manual.

File Menu

The File menu allows you to load and save full and partial screen images in two formats. The File menu also lets you print the screen, execute a DOS shell, and exit the Graphics QuickScreen program.

- **New Screen**

  The New Screen option is used when you wish to start a new screen or change to a different screen mode. You may select from EGA 16-color

640x350 resolution with either 25 or 43 text rows or VGA 16-color 640x480 resolution with either 30 or 60 text rows.

If a screen is currently being edited and has been altered since it was last saved, Graphics QuickScreen will prompt you to save before continuing. Once the screen is successfully saved, the screen will be cleared to the current background color. (The background color is set from the **System** dialog box under the **Settings** menu.) All previously defined fields are cleared from memory. The color palette will not be reset unless you change to a different screen mode. This lets you use the color palette from any other screen.

You can optionally display a grid of dots corresponding to the internal ROM text height and width for the selected screen mode by checking the **Show Grid** check box on the **System** dialog box before selecting **New Screen**. This grid is useful because it helps you to visualize your forms while they are being designed. Some monitors will occasionally lose the mouse cursor after **New Screen...** is selected. In this case, the cursor can be restored by pressing **Ctrl-F1**.

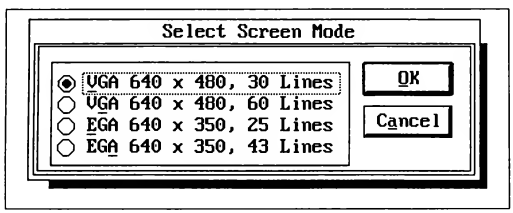


Figure 7: New Screen Dialog Box

- **Open...**

The Open selection is used to retrieve a screen which has already been designed and saved on disk. The required extension for Graphics QuickScreen screens in the file selection dialog box is .PCX. Any screen which had been saved using the Graphics QuickScreen **(File) Save...** command, **(File) Save As...** command, QS2GQS conversion utility, or any other high resolution EGA- or VGA-compatible .PCX file may be loaded and displayed using this option.

If the screen being loaded was saved by Graphics QuickScreen in a different screen mode, Graphics QuickScreen will ask you if you want to change to the appropriate mode. If you select No, the .PCX file will be displayed in the current screen mode. Some monitors will occasionally

lose the mouse cursor after a new screen is loaded. In this case, the cursor can be restored by pressing **Ctrl-F1**.

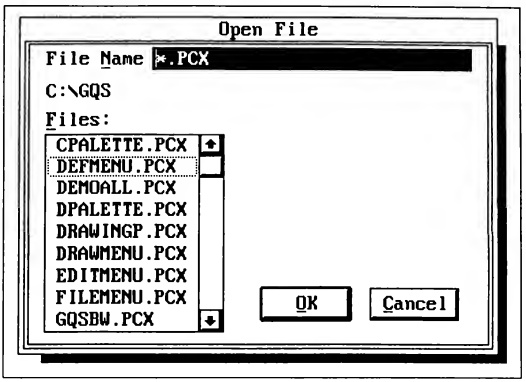


Figure 8: Open File Dialog Box

- **Save...**

The Save option saves the screen currently being edited and creates a file with the extension .PCX. (Graphics QuickScreen saves full or partial graphic screens as .PCX files. This format uses a data compression algorithm to minimize the amount of disk space used to store graphic images.) If this is the first time a screen is being saved, a dialog box will appear prompting you for a file name. This dialog box is shown in Figure 9.

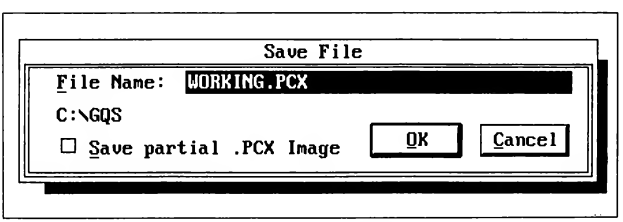


Figure 9: Save File Dialog Box

If you wish to save only a portion of the screen you can check the **Save Partial .PCX Image** check box. Upon exiting the dialog box, you will be prompted to identify the region to be saved. A drawing cursor will appear as soon as you move the cursor allowing you to define the region by drawing a box around it. (Note that partial .PCX images can only be saved and restored using a mixed coordinate system of text columns and pixel rows.)

If you have defined any fields for your screen, a Form (.FRM) file with the same base name as your screen is also created. The .FRM file contains information about the various fields that you have defined such as their location, field name, field type, and so on. You may also save this information in a BASIC source module that can be called from your program by checking the **Create BASIC .FRM file** box. (The .FRM file will still be created as it is needed later to load the form back into the Graphics QuickScreen editor.)

If a file name has already been assigned to your screen, selecting **Save** will save the screen without prompting you for a new file name.

.PCX files employ a 128 byte header. This header contains information about the size of the image, number of color planes, bits per pixel, and palette information. Of the 128 bytes, only the first 67 are needed. Graphics QuickScreen uses several bytes in the unused portion of the header to store the screen mode and the height in pixels of a standard text character for the current screen mode. This information tells Graphics QuickScreen and the ShowForm subroutine the screen mode and number of text rows to set. (Although it is possible to determine the correct screen mode from the original .PCX file header, it is not possible to determine the number of screen rows.)

If a partial .PCX file is saved, the upper left corner coordinates of the image are saved into the header as well. There is no recognized standard for partial .PCX image coordinates, so these bytes are unique to .PCX screens saved by Graphics QuickScreen.

This data is placed in the .PCX header starting at byte 85 and is assigned as follows:

Byte	Data
85	The current screen mode as indicated by GPDat%(31). GPDat%(31) is assigned a value of 5 for EGA 640x350 screens, and 8 for VGA 640x480 screens.
86	The height in pixels for a standard text character in the current screen mode. This value along with the screen mode lets Graphics QuickScreen and the ShowForm subroutine determine the correct number of screen rows.
87	The upper-left pixel row of the saved image (partial .PCX images only)
88	The upper-left text column of the saved image (partial .PCX images only)

Table 7: Graphics QuickScreen .PCX header bytes

- **Save As...**

  Save As is similar to **Save** and it differs only in that a dialog box is always presented allowing you to enter a new file name. **Save As**... is useful if you have changed a screen and want to save it under a different name to avoid overwriting the original version.

- **Save Paste Buffer...**

  This option lets you save the current contents of the paste buffer as a bit-mapped image with a .GMP (**G**raphics bit **MaP**) extension. Images are placed in the paste buffer by using the **Move** or **Copy** commands selected from the **Edit** menu or from the Drawing Palette. The paste buffer is limited to storing graphic images 64k or smaller and thus determines the maximum size for the saved image.

  There are several advantages when using the .GMP format for saving small screen images. Bitmapped images are not restrained by the mixed coordinate system of text columns and pixel rows and can therefore be easily restored to any X/Y pixel coordinate. They can also be displayed more rapidly than partial .PCX images because they do not have to be decompressed as they display. They can also be recalled into the Graphics QuickScreen editor and easily pasted onto other screens.

  Disadvantages are that the .GMP format requires more memory to store a given image than the .PCX format, and that .GMP images do not contain any palette information.

This method is best suited for displaying small graphic images very quickly. (.GMP images are used to display some of the more elaborate tool icons on the Drawing Palette.)

- **Load Paste Buffer...**

  This option retrieves images that were saved with **Save Paste Buffer...** option and it stores them in the paste buffer. These images can then be placed anywhere on the screen by selecting the **Paste** option of the **Edit** menu.

- **Print Screen...**

  This option takes a "snapshot" of the current screen and sends it to either a 9-pin Epson compatible dot matrix or a Hewlett-Packard compatible laser printer. The default printer port is set to 1, but may be changed from the **System** dialog box under the **Settings** menu. A dialog box will appear prompting you for the following information:

Dot Matrix or Laser	Select whichever is appropriate for your printer.
Portrait Mode	Images will be printed in landscape mode (sideways) unless this option is checked.
Tiled Colors	With this option checked, colors will be translated to representative tile patterns. If unchecked, all colors are printed as solid black and black portions of the screen are not printed.
Swap Colors	With this option checked, white will be printed as black and black will not be printed.
Laser Scaling	This feature applies only to Laser printers and determines the size of the printout. 75 DPI produces the largest image while 300 DPI produces the smallest. (Acceptable values are 75, 100, 150, and 300.)

Note that this option gives you a dot-for-dot reproduction of what appears on the screen. Since the resolution for a video monitor and most printers are different, the aspect ratio of the printed image may be somewhat distorted. This is particularly true of EGA adapters at 640x350 resolution. (The pictures shown in this manual were created with the Graphics QuickScreen editor on a VGA display and printed on a laser printer at either 100 or 150 DPI.)

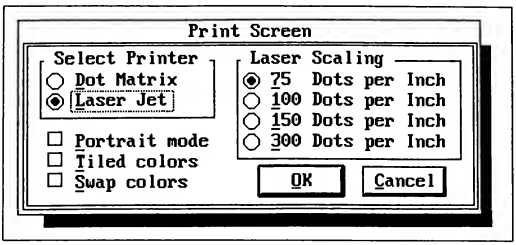


Figure 10: Print Screen Dialog Box

- **DOS Shell**

  The DOS Shell lets you perform DOS functions while Graphics QuickScreen remains in memory. In order for this feature to work properly, the file COMMAND.COM should reside on the root directory of your boot drive. You may specify where COMMAND.COM resides by using the COMSPEC command in your AUTOEXEC.BAT file. For example, if COMMAND.COM is on C:\DOS you may place the following command in the AUTOEXEC.BAT file:

  ```
  COMSPEC=C:\DOS
  ```

  As always, if you change the COMSPEC setting using AUTOEXEC.BAT, you must either run it again or reboot your computer before the change will take effect.

- **Exit**

  Exit ends your session with Graphics QuickScreen. If the current screen has been changed since it was last saved, then Graphics QuickScreen will prompt you to save the screen before exiting. (Ending the program also creates a configuration file in the current directory. This file is named "GQS.CNF" and stores the current path for icon, font, and tile files as well as several other user settings such as the selected printer port, current background color, mouse sensitivity, and so on.

Edit Menu

The Edit menu lets you access a number of Graphics QuickScreen's graphic editing tools.

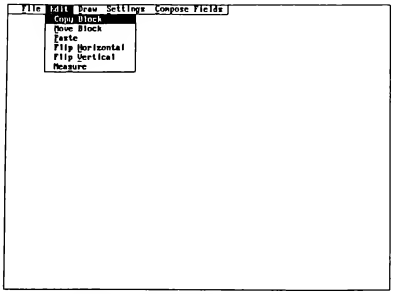


Figure 11: The Edit Menu

- **Copy Block**

  This option allows you to capture an image from your screen and make one or more copies of it anywhere you like. See *Copy/Move/Paste* under *The Drawing Palette* section of this manual for more details.

- **Move Block**

  This option allows you to capture an image from your screen and move it to a new location. The background color defined in the Settings menu is used to replace the image once it has been moved. See *Copy/Move/Paste* under *The Drawing Palette* section of this manual for more details.

- **Paste**

  This option recalls the image that is stored in the paste buffer. When selected, the image will appear in the upper left-hand corner of the screen and it may be placed anywhere on the screen as if you had captured it with the **Copy** procedure.

- **Flip Horizontal/Vertical**

  The **Flip Horizontal** or **Flip Vertical** choices let you select any rectangular portion of the screen and graphically flip it horizontally or vertically.

  → To Flip a region:

  1. Select **Horizontal** or **Vertical Flip** from the **Edit** pull-down menu. A drawing cursor will appear.

  2. Draw a box surrounding the region to be flipped, using the box drawing procedure.

  3. The image will be flipped as soon as you click the left mouse button.

Note that even though the horizontal and vertical flipping algorithms are very similar, flipping an image horizontally is considerably slower than flipping it vertically because of the way video memory is organized.

- **Measure**

The **Measure** option allows you to measure distances by row and column or by X and Y pixels. Measurements are taken from the last point clicked. Distances are displayed in the **Status Box**.

→ To take a measurement:

1. Select **Measure** from the **Edit** pull-down menu. The **Status Box** will appear if it is not already active. A drawing cursor will also appear.

2. Click the mouse on whatever starting point you desire. The **Status Box** will display the distance as you move the mouse cursor.

Pressing the **T** key will toggle between **T**ext (row and column) and pixel (x and y) coordinates.

Draw Menu

The Draw Menu is used to access additional drawing tools not found on the Drawing Palette.

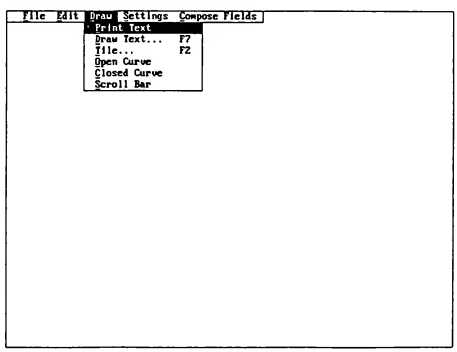


Figure 12: The Draw Menu

- **Print Text**

  This option allows you to type text onto the screen using standard row and column coordinates. This routine uses the same ROM font used by your computer but prints without disturbing the underlying screen. See *Print Text* under *The Drawing Palette* section in this manual for more details on using this option.

- **Draw Text**

  This option is used to print text in a selected font anywhere on the screen at any angle and in any color. The text may optionally be scaled and/or italicized.

  → To draw text:

  1. Select **Draw Text** from the **Draw** pull-down menu and a dialog box will appear.

  2. The dialog box prompts you for the following information:

     - Text: Enter the text to be printed, up to 10 lines.

     - Font: Select the desired font. Five pre-defined fonts are available, or you can select from fonts that you have created using the GPFont editor utility program*.

     - Scale: Enter the scaling factor. The range for scaling is from approximately .75 to 50, and can be any fractional value in between. A value of 1 produces a font in the point size specified. Values less than .75 are acceptable but generally produce poor results. As the font is scaled up the apparent resolution goes down. This condition is inherent in almost any scaled font.

     - Text Angle: Enter the desired angle at which to draw text. A value of 0 draws text horizontally. Positive angles rotate the text clockwise while negative angles rotate the text counterclockwise.

     - Italics Angle: Enter the desired italics angle. A value of 90 produces normal text, while values between 60 and 70 degrees produce typical italicized text. Virtually any angle can be used and you can even slant text backwards by using angles greater than 90 degrees. Note that if a text angle other than 0 is specified, italics is disabled.

  3. When all the information is correct click the OK button. A rectangle will appear indicating the size of the text to be printed

(not counting decenders). If a text angle is specified, a drawing cursor will appear instead.

4. Position the rectangle/drawing cursor where you would like to place the text and click the left mouse button. The drawing cursor positions the upper left corner of the text. Text color can be changed by pressing the number keys. If you make a mistake you can clear the text by pressing **F10**.

To create bold fonts, print the text as usual but do not move the cursor. With grid snap off, press the **Right** or **Left** arrow key to move the cursor one pixel to the right or left, and then press **Enter**. The text will be drawn again offset by one pixel creating a bold effect. Shadowed or embossed text can be created by changing colors before drawing the text a second time.

* If you also have Crescent's Graphics Workshop or GraphPak Professional, you can create additional fonts using the font editor supplied with them. Save the fonts using the names User1, User2 and so forth. as found on the **Draw Text** dialog box. Place them in your Graphics QuickScreen directory and they may then be accessed from the **Draw Text** dialog box as well. The following font files are provided:

HELV8.GFN	8 point helvetica
HELV12.GFN	12 point helvetica
TROM12.GFN	12 point Times Roman font
FUTURA.GFN	14 point Futura font
OLDENG.GFN	14 point Old English Font

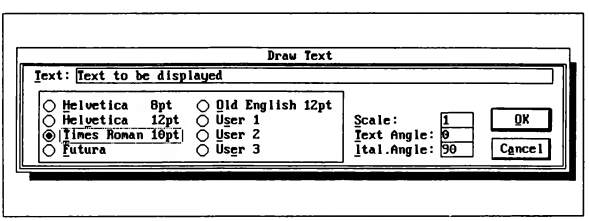


Figure 13: Draw Text Dialog Box

The drive and directory for these files can be specified in the **Set Paths** dialog box. The default location is the current directory unless the configuration file (GQS.CNF) indicates otherwise.

■ Tile

The Tile palette gives you access to an additional 119 dithered colors and 28 different tile patterns using only the 16-color palette.

A dithered color is produced by selecting any two colors and alternating them every other pixel to produce a different apparent color. This is accomplished by using a technique known as "tiling", where a 16x16 pixel pattern is defined and then repeated much like tiles on a wall. These tiles are generally used as fill colors when 16 colors are inadequate, or to produce interesting backgrounds for your screens.

The tiling procedure works just like the **Flood Fill** option on the Drawing Palette, and it will fill an entire area contained within a single color. (See *Flood Fill* under *The Drawing Palette* section for more details.)

→ To tile a region:

1. Select **Tile** from the **Drawing** pulldown menu. The **Tile Palette** will appear at its last used location. The **Tile Palette** can be placed anywhere on the screen by clicking the bar at the bottom of the palette and dragging it to the desired position.

2. Select a tile by clicking on it with the left mouse button. The **Tile Palette** will disappear. Paint the region by pointing and clicking the mouse as you would for the **Flood Fill** procedure.

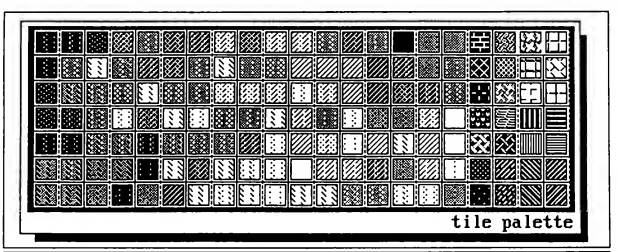


Figure 14: Tile Palette

You may continue painting with the current tile, or you may return to the **Tile Palette** by clicking the right mouse button. One more right click will return you to the Drawing Palette.

The **Tile Palette** image is stored as a bit map in the TILEPAL.GM4 file. The tiles are stored in the file TPAL.TIL. These files should reside in the current directory or in the directory specified in the **Set Paths** dialog box.

■ Open/Closed Curve

The **Open/Closed Curve** options let you easily draw smooth curves of almost any shape.

→ To draw an open/closed curve

1. Select **Open Curve** or **Closed Curve** from the **Draw** menu and a drawing cursor will appear.

2. Draw lines that roughly define the shape of the curve, similar to the line drawing procedure. You are allowed up to 100 line segments to define the shape.

3. Click the right mouse button when you have completed the rough outline to end line drawing. If you change your mind or make a mistake when defining the outline, click the right mouse button again to delete the line segments and start a new curve. If you are satisfied with the shape, click the left mouse button and the line segments will disappear. A hyperbolic smoothing algorithm is applied to the shape you have defined, and a "smoothed" version of your outline will be drawn.

An open curve differs from a closed curve only in that the endpoint will be smoothly joined to the starting point. This can be an important difference depending on the desired result.

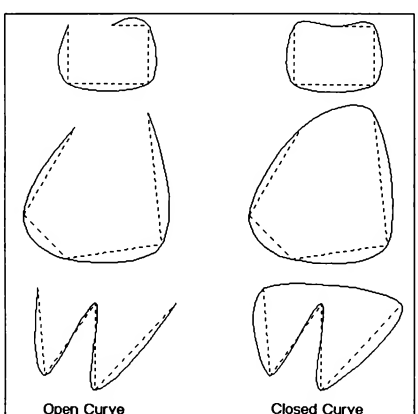


Figure 15: Effect of Smoothing Algorithms

You may continue to draw smoothed lines or you may return to the Drawing Palette by clicking the right mouse button.

- **Horizontal/Vertical Scroll Bars**

Scroll bars provide an easy way for users to enter a number from a range of values in your programs. Like push buttons, they can be drawn in any color but require three shades of the same color for the best 3D effect. See *Push Button* under the *Drawing Palette* section of this manual for a more detailed description on using colors. The sliding range of the scroll bar can be changed to any other color as well.

→ To draw a vertical or horizontal scroll bar:

1. Select **Scroll Bar** from the **Draw** pulldown menu. A drawing cursor will appear.

2. Place the cursor at any corner where you wish to start the scroll bar and then click the left mouse button.

3. Move the cursor until the box is the size desired and click the left mouse button. If the box is wider than it is high, a horizontal scroll bar is drawn. If the box is higher than it is wide, a vertical scroll bar is drawn. The scroll bar will be drawn in the selected color.

4. If you wish to use a different color for the sliding portion of the scroll bar, use the **Flood Fill** routine from the Drawing Palette to paint the new color.

You may continue to draw scroll bars or return to the Drawing Palette by clicking the right mouse button.

Note that at this point the scroll bar is just a graphic image, and needs to be defined as a field* before it will actually function as a scroll bar.

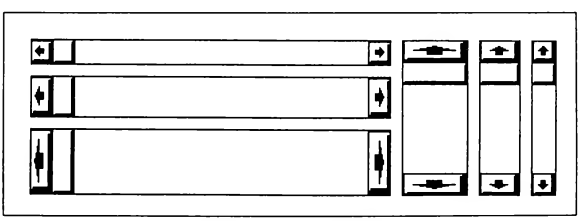


Figure 16: Scroll Bars

Scroll bars may be almost any size but the length should be no less than 49 pixels to allow room for the two push buttons and the scroll pointer.

* We recommend that you draw scroll bars with grid snap on. This greatly simplifies the process of defining them later as fields.

Settings Menu

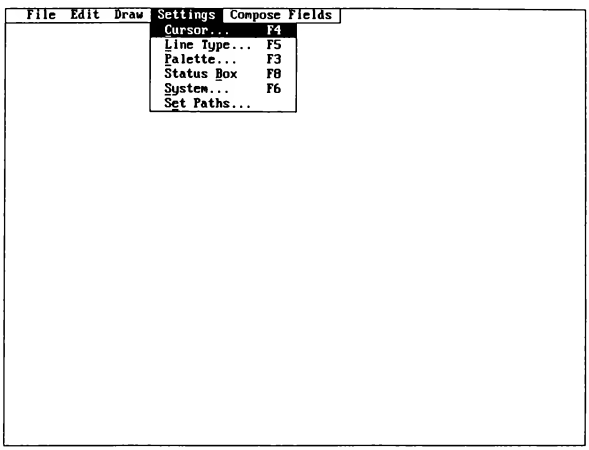


Figure 17: The Settings Menu

The Settings menu allows you to set several global parameters that affect how the drawing and editing routines operate.

- **Cursor...**

This option lets you pick the style of cursor to use during drawing and editing operations.

→ To select a drawing/editing cursor:

1. Select **Cursor...** from the **Settings** pulldown menu. A dialog box will appear allowing you to pick one of four cursor styles: Crosshair, Full Crosshair, XCrosshair or Square.

The Crosshair is the default drawing/editing cursor. The Full Crosshair extends to the full height and width of the screen and can help make it

easy to line one object up with another. You will notice that when drawing vertical or horizontal lines with either the Crosshair or the Full Crosshair that the part of the cursor that is over the "rubberband" line appears in a different color. It also changes as the background color changes.

This is a side effect of the XORing technique used to draw both the line and the cursor. This will rarely cause a problem and is often useful when trying to locate the cursor precisely at an edge where two different colors meet. If for some reason this is a problem, you can use the X Crosshair instead. The square cursor is primarily used for the paintbrush routine, but it can be used for drawing as well. Its size will correspond to the Brush Size specified in the **Settings** dialog box.

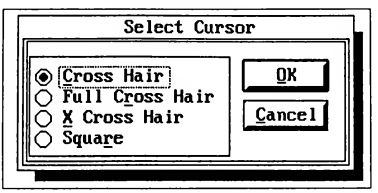


Figure 18: Cursor Type Dialog Box

■ Line Type

The **Line Type** dialog box selects the line style to be used to draw lines, boxes and polygons. You can select from seven pre-defined line types or define your own custom line style. Custom line types are created by defining a 16-bit line mask where each bit corresponds to a pixel on the screen. A value of one will plot a point while a value of zero will not. This pattern is then repeated every sixteen pixels for the length of the line. To create a simple dashed line, select the *Custom Mask* option button and enter a text string such as:

```
1111110000111111
```

or

```
1111111100000000
```

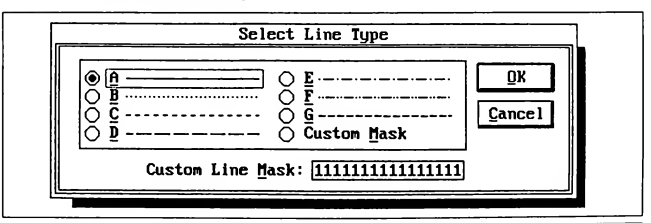


Figure 19: Drawing Line Type Dialog Box

All subsequent line, box and polygon drawing will be performed using the selected line type. You can easily toggle between the selected line type and solid line drawing by pressing the backslash (\|) key. A single beep indicates that solid line drawing is in effect, while two beeps indicate that a broken line type is in effect.

Palette

The **Color Palette** lets you assign any of the EGA palette's 64 colors or any of the VGA palette's 256k colors to any of the 16 drawing palette colors. Use of the proper colors can greatly enhance the appearance of a well designed screen.

→ To select or edit a color:

1. Select **Palette** from the **Settings** pulldown menu. Either an EGA or VGA palette editor will appear, depending on the current screen mode. The **Palette Editor** can be placed anywhere on the screen by clicking the bar at the bottom of the editor and dragging it to the desired position.

2. Select the color to change or edit by clicking on it with the mouse, or by pressing the corresponding numeric key. The selected color will appear in the larger window on the right side of the editor.

3. Cycle through the 64 EGA colors by using the scroll bar at the bottom of the editor. The new color will be assigned to whichever color you select.

The VGA palette editor allows you to mix your own colors by providing three scroll bars to control the red, green, and blue components of the color. These values can range from 0 (off) to 63 (full intensity). The selected color will be assigned to whatever color you create. When controlled from the keyboard, scroll bars are accessed by pressing **Tab** or **Shift-Tab** and are controlled using the cursor direction keys.

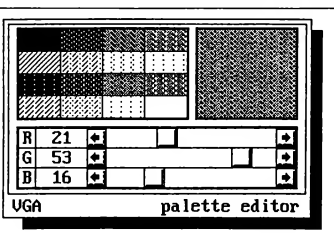


Figure 20: Color Palette

- **Status Box**

  The **Status Box** can be used to display the current setting of several different drawing and editing parameters. It can be placed anywhere on the screen by clicking on it with the mouse and then dragging it to the desired position as long as no other pop-ups are active.

  The **Status Box** displays the following parameters:

  1. Current drawing color
  2. Mouse cursor position

     You can toggle between standard row and column coordinates, or X and Y pixel coordinates by pressing the **T** (Text) key.

  3. Snap status (on/off)

     Coordinate labels are shown in upper case if snap is on or lower case if snap is off. You can toggle snap on and off with the S (Snap) key.

  4. Relative Mode (on/off)

     When on, cursor coordinates are displayed relative to the last position clicked. A CHR$(7) Dot will appear in the Status Box just to the left of the row or Y label. Coordinates are relative to the upper left corner of the screen when Relative Mode is off. Row and column coordinates start at 1, 1 while X and Y pixel coordinates start at 0, 0. You can toggle the status on and off with the **R** (Relative) key.

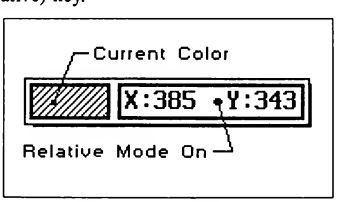


Figure 21: Status Box

- **System**

  The **System** option allows you to set several default settings that affect how specific drawing and editing procedures operate.

  → To change a system setting:

  1. Select the **System...** option from the **Settings** pulldown menu. A dialog box will appear prompting you for the following information:

- **Block Options:**

  Copy/Move Block — Determines whether copy or move is active when the Copy/Move icon is selected from the Drawing Palette. The default setting is for copy.

  BG Color — Specifies the background color (0 - 15) to use when replacing an area that has been moved. The default background color is black (0).

  XOR on — Selects whether or not the pasted image will be placed as either an XORed image or as an opaque image. With XOR on, any black portion of the copied or moved image will appear transparent when pasted. All other colors in the pasted image will be XORed with the underlying screen as well. This will cause any color other than black or white to vary with the underlying screen colors. The default is set to XOR off.

- **Status Display:**

  Text/Pixel — Selects whether the **Status Box** will display screen coordinates in text rows and columns or in X/Y pixels. This option can also be toggled from the keyboard by pressing the **T** key whenever the **Status Box** is active.

  Status On — When checked, the **Status Box** is displayed.

- **Snap Settings:**

  X Snap — Set the desired X Grid snap spacing (0 - 99) in pixels. When painting fields this setting should be set to 9.

  Y Snap — Set the desired Y Grid Snap spacing (0 - 99) in pixels. When painting fields this value should be set to your screen modes text height (in pixels) plus 1.

  Snap On/Off — When checked, Grid Snap will be on. Grid Snap can also be toggled on and off at any time during drawing or editing by pressing the S key. The default is Grid Snap On.

- **Pixel Grid On** — This option determines whether or not grid lines will be displayed in a zoomed image. Generally, grid lines are an aid when editing an enlarged image. If they are not required uncheck this option.

- **Show Grid** — When checked, this option causes a dot grid corresponding to the size of a standard text character to appear whenever **New Screen...** is selected from the **File** menu. The default setting is off.

- **Clear on Delete** — When checked, fields will be cleared to the current background color when they are deleted during the **Enter Field definitions** procedure.

- **Corner Radius** — Sets the desired corner radius in pixels to be used with the radius box drawing procedure. Values can range from 1 to 999 pixels. The default value is 15.

- **Brush Size** — Determines the size in pixels of the paintbrush. This setting also controls the size of the square drawing cursor.

- **Mouse Sens.** — Determines the sensitivity of the mouse cursor. This setting can range from 1 to 99, but its useful range is from 1 to about 30. Values above 30 or so are hopelessly insensitive. As a reference, the mouse sensitivity setting within the QuickBASIC editor defaults to about 10 while the Graphics QuickScreen default is set to 4.

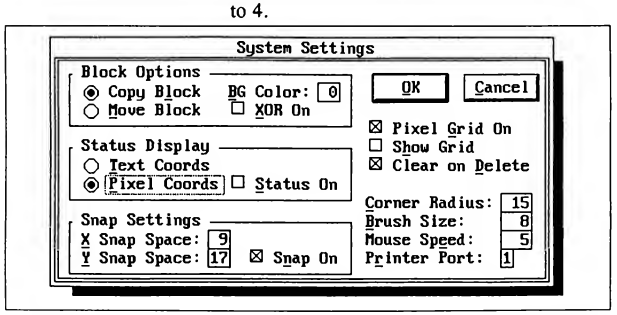


Figure 22: Systems Dialog Box

Set Paths

The **Set Paths** option lets you specify the new drive and directory where the various Graphic QuickScreen support files are located. These files are the Drawing Palette icon files (SCRIBBLE.GMP, PBRUSH.GMP, BUCKET.GMP, and CLRWHEEL.GMP), the Tile Palette bit-map and its related tile file (TILEPAL.GM4, TPAL.TIL), and the font files used by the Draw Text option (files with a .GFN extension). This lets you run GQS from any other directory and still have access to the required support files.

The new path should be entered without a trailing backslash:

```
C:\GQS\FONTFILE
C:\GQS\MISC
```

The new paths are stored in the GQS.CNF configuration file that is created whenever you end the program. This file is saved in the current directory and loaded automatically whenever GQS is run again from the same directory.

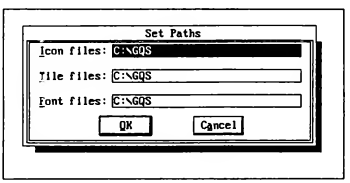


Figure 22a: Set Paths Dialog Box

Compose Fields Menu

Graphics QuickScreen has the ability to manage a large group of predefined fields. This feature allows data entry screens to be quickly and easily developed.

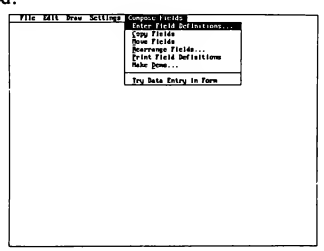


Figure 23: The Compose Fields Menu

- **Enter Field Definitions...**

  This command defines fields to be added to the current screen thereby creating what ultimately is to be a complete form. Each field may be defined to accept very specific information. For example strings, numbers, or dates.

- **Move Fields**

  This selection allows you to move a field or range of fields anywhere on the screen. Fields are moved using a procedure very similar to the graphic **Move Block** procedure found on the Drawing Palette or under the **Edit** menu. **Move Fields** differs from **Move Block** in that if the area selected contains previously defined fields, the field definitions are moved along with their graphic images.

  Only those fields whose four corners fall completely inside the encompassing box will be moved. Moved images are replaced with the current background color. The background color is set in the **System** dialog box found under the **Settings** menu.

  You may use any Grid Snap settings to identify and move the region. However, if the region contains any text entry fields, Graphics QuickScreen will limit movement such that the field can only be placed at standard text rows and columns. Mouse fields, push buttons and scroll bars can be moved to any pixel location. Text fields cannot be moved to the bottom screen row.

- **Copy Fields**

  This selection copies a field or range of fields to eliminate the need to manually type in duplicated information. Fields are copied using a procedure very similar to the graphic **Copy Block** procedure found on the Drawing Palette or under the **Edit** menu. **Copy Fields** differs from **Copy Block** in that if the area selected contains previously defined fields, the field definitions are copied and moved along with their graphic images. Unique field names are automatically created for each new field. Only those fields whose four corners fall completely inside the encompassing box will be copied.

  You may use any Grid Snap settings to identify and move the region. However, if the region contains any text entry fields, Graphics QuickScreen will limit movement such that the field can only be placed at standard text rows and columns. Mouse fields, push buttons and scroll bars can be copied and placed at any pixel location. Text fields cannot be copied to the bottom row of the screen.

This feature provides an extremely fast way to generate new fields without going through the entire field definition process.

- **Rearrange Data Fields...**

  This selection allows previously-defined data fields to be moved from their data entry order (when **Enter** or **Tab** are pressed).

- **Print Field Definitions...**

  This option creates a printed listing of all field definitions for the current form and thus serves as a handy documentation utility. This printout information is sent to any printer port. The printer port can be assigned from the **System** dialog box under the **Settings** menu. The default port is LPT1.

- **Make Demo...**

  This selection creates write a BASIC source file that you can use as a starting point to help develop your own source code. The code produced will run as if **Try Data Entry in Form** had been selected.

  If push buttons or scroll bars have been defined, appropriate SELECT CASE statements will also be generated so that you need only write code that responds to the values that they return. If multiple choice fields have been defined, the Choice$() array will be set up to provide temporary choices for the list boxes. You will have to modify this code to contain the actual choices your program requires. (See *Setting Up Multiple-Choice Fields* for more information.)

  When you select **Make Demo...** a dialog box will appear prompting you for the following information:

Use .FRM File	This option causes the form definition file to be loaded from disk. This provides the smallest .EXE size but requires that the .FRM file be distributed along with your program.
Use BASIC module	This option creates and assigns the form definitions from a BASIC module. This allows you to include the form definition file directly into your final .EXE program but results in a somewhat larger program compared to loading the field definitions from disk.
Demo name	Enter the desired name of the demo without a path or extension. The name will default to the word *DEMO* plus the first four letters of the form name.

When you are satisfied with the demo settings, click OK to generate the BASIC code. A .MAK file that includes all of the required modules for your form will be created, as will a main module that calls up your screen and form definition file. These files will be placed in the current working directory. To run the demo, follow these steps:

1. Save your form and exit Graphics QuickScreen

2. Log on to your working directory and start BASIC with the appropriate library:

   ```
   QB /L GFORMS  (for QB4.0 through BASIC 6.0)
   ```
   or
   ```
   QBX /L GFORMS7  (for BASIC 7.0 or later versions)
   ```

3. Check the DEMONAME.MAK file to make sure all of the required modules and the form's .PCX file are in the current directory. If you work from the Graphics QuickScreen directory, the required files will already be there. If you work from any other directory the required modules will have to be copied into your working directory before the demo will run.

4. Load the demo using the **Open** command from the **File** pulldown menu.

5. Once loaded, the demo can be run by pressing **Shift-F5**

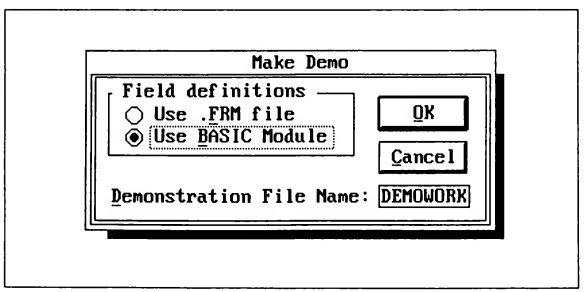


Figure 24: Make Demo Dialog Box

■ **Try Data Entry in Form**

The Graphics QuickScreen editor lets you test how a form will operate when run from QuickBASIC. After choosing this command, the Graphics QuickScreen environment will allow you to enter data in any field of the form. When you have finished, simply press **Esc** in order to return to the screen editing mode.

Function Keys

Function keys can be used to execute a number of the commonly used menu commands. Their functions are listed in the table below:

Key	Action
F1	Display a help screen listing the various special key and function key assignments
F2	Display the Tile Palette
F3	Display the Palette Editor
F4	Display the Select Cursor dialog box
F5	Display the Line Type dialog box
F6	Display the System Settings dialog box
F7	Display the Draw Text dialog box
F8	Toggles the Status Box on and off. May be accessed anytime during drawing or editing
F9	Display current memory status
F10	Undo any drawing or editing performed since the last time the Drawing Palette or PullDown menu was invoked

Table 8: The Function Keys

Special Keys

Several keys have been defined to serve as toggles for changing the status of the various drawing and editing parameters. These keys and their functions are listed in the table below:

Key	Action
S	Toggles Grid **S**nap on and off. This key can be used during any drawing or editing operation. Note that when Grid Snap is on, labels in the status box are displayed in upper case.
T	Toggles the Status Box display between **T**ext (Row/Column) coordinates and pixel (X/Y) coordinates.
R	Toggles the Status Box display between **R**elative and absolute screen coordinates. Absolute coordinates are measured from the upper left-hand corner of the screen. Pixel coordinates start at 0, 0 while text coordinates start at 1, 1. Relative coordinates are measured from the last point selected.
\ !	Toggles the line type for drawing lines, boxes and polygons between solid and the previously selected line type. A single beep indicates that solid line drawing is in effect while two beeps indicate that the previously selected line type is active.
A	Toggles **A**rc rotation. Arcs can be drawn either clockwise or counterclockwise from the selected starting point to the selected end point. Assuming a black background, the drawing cursor will appear white to indicate counterclockwise rotation or yellow to indicate clockwise rotation. This key is active only during the Arc drawing procedure.
F	Toggles between the current Grid Snap settings and Grid Snap settings that correspond to the current text size. A black box "■" will appear in the **Status Box** when snap coordinates correspond to the current text size.

Table 9: Special Keys

Fields

Graphics QuickScreen has the ability to create fields which gather user input when a form is used from BASIC. A screen with field definitions is called a form. When using forms from BASIC, field information may be passed to and from a calling program.

In order to understand the use of fields in Graphics QuickScreen we will first present the many field types available. Next, the Field Settings dialog box is discussed, since each field may be customized to a certain extent. The next section discusses the power in using numeric formulas. And last, the entire process of defining fields is described.

Field Types

The field type describes the data which is to be entered at a particular field. For example, there is a Social Security Number field type which accepts numerical information only in the form ###-##-####. Graphics QuickScreen contains built-in logic for each field type, making additional formatting or syntax-checking by the calling program unnecessary. Graphics QuickScreen also supports four additional field types that do not accept data but must be defined as fields for them to function. These are: mouse fields, push buttons, horizontal and vertical scroll bars.

Certain field types are fixed-length and generate mask characters. These are simply delimiting characters such as the dashes in a social security number that help to format a field on the screen. Graphics QuickScreen inserts mask characters for you and skips over them automatically when the field is being used in a form.

Following is a description for each field type available in Graphics QuickScreen.

- **String**

  Alphanumeric characters, both upper-case and lower-case are accepted. This field is useful for collecting any general single-line string information.

- **Proper String**

  Alphanumeric characters are accepted and the first letter in each word is automatically capitalized during data entry. This field is useful for names and addresses.

- **Upper Case String**

  Alphanumeric characters are accepted, and all characters are automatically converted to upper-case during data entry. This field is useful for abbreviations, state codes and part numbers.

- **Numeric**

  Numeric characters are accepted but are treated as strings. This field is useful for telephone numbers and zip codes.

- **Scrolling Text**

  Lines of text longer than the actual defined text window may be entered. You may also specify what type of text is to be accepted from 5 options:

  1. All characters
  2. Integer characters only (0123456789)
  3. Single or Double precision characters only (01234567890+- ,.ED)
  4. All letters are capitalized
  5. Proper strings where the first letter of each word is capitalized

  See the documentation for SCROLLIN.BAS for more information on selecting the desired option.

- **Multi Line Text**

  Several lines of alphanumeric characters are accepted. This field is ideal for notes.

- **Logical**

  A pre-defined "true" or "false" character is accepted. This field is therefore useful for yes/no or check/uncheck fields.

- **Integer**

  An integer number in the default range -32768 to 32767 is accepted. This field is useful for entering an integer value such as the quantity of items in a sale.

- **Long Integer**

  A long-integer number in the default range -2,147,483,648 to 2,147,483,647 is accepted. This field is useful for entering a very large integer number.

- **Single Precision**

  A single-precision number in the default range 3.402823 E+38 to 1.401298 E-45 is accepted. This field is useful for general decimal numbers.

- **Double Precision**

  A double-precision number in the default range 1.7976931 D+308 to 4.940656 D-324 is accepted.

- **Currency**

  An amount of money in the double-precision range is accepted. The currency symbol for the field may be defined so that dollars, yen, or other currencies may be used. Note that this field does not use the Currency data type.

- **Date MM-DD-YYYY**

  A date in the default range 01-01-1900 to 01-01-2065 and in the American format (MM-DD-YYYY) is accepted. The dash (-) mask character is added automatically.

- **Date DD-MM-YYYY**

  A date in the default range 01-01-1900 to 01-01-2065 and in the European format (DD-MM-YYYY) is accepted. The dash (-) mask character is added automatically.

- **Phone Number**

  A phone number in the form *(###) ###-####* is accepted. The parentheses and dash mask characters are added automatically.

- **Zip Code**

  A zip code in the form *#####-####* is accepted. The dash mask character is added automatically.

- **Social Security Number**

  A social security number in the form *###-##-####* is accepted. The dash mask characters are added automatically.

- **Relational**

  Allows you to specify what file and which field in that file is to be used for the current field. Although Graphics QuickScreen does not process these fields automatically, relational fields allow a calling program to access data in other form files.

- **Multiple-Choice Array**

  Presents a vertical menu of choices. This menu is defined in a string array by the calling program and then displayed whenever the field is accessed. Menu colors are set in the GPDat%() array.

- **Mouse Field**

  Mouse fields allow you to define rectangular regions on the screen that will return a single user-defined key code whenever they are selected in a form.

  A mouse field is activated by clicking on it with a mouse, moving to the field and pressing **Enter**, or pressing the key which has been assigned to

it. The field can be outlined when it is the current field and can also be highlighted in any color when activated. The field may also be set up to function as a toggle such that each selection highlights or un-highlights the field.

When defined as a toggle, a mouse field will occupy one byte in a data file. Otherwise, a mouse field contains no data and will not occupy space in a data file.

- **Push Button**

  Push buttons return a single user-defined key code whenever they are selected in a form. Common example push buttons are OK and Cancel, which would typically return the key codes for **Enter** and **Esc**, respectively. Although the button fields may show any words or pictures and return any keycode you choose, they always return a single value.

  A push button is activated by clicking on it with a mouse, moving to the field and pressing **Enter**, or pressing the key which has been assigned to it.

  Push buttons do not have any data associated with them and thus do not occupy space in a data file.

- **Horizontal/Vertical Scroll Bars**

  Scroll bars provide an easy way for users to enter a number from a range of values in your programs. The numbers that they may represent can range anywhere from -32768 to 32767 and can be returned in any step increment. See *Scroll Bars* for more detailed information.

  Table 10 presents a summary of Graphics QuickScreen's twenty-three field types. The field type names appear in the **Field Types** dialog box shown in Figure 25 and are encountered when defining fields using the steps outlined under the section *Creating Data Entry Fields*.

• String	
String	Alphanumeric characters
Proper String	Alphanumeric characters; the first letter of each word will be capitalized
Upper case String	Alphanumeric characters; each alphabetic character will be capitalized
Numeric	Numeric characters only
Multi Line Text	Alphanumeric characters; several lines of text may be entered and edited
Scrolling Text	Single line text that may be scrolled left or right
• Numeric	
Integer	-32768 to 32767
Long Integer	-2,147,483,648 to 2,147,483,647
Single Precision	3.402823 E+38 to 1,401298 E-5
Double Precision	1.7976931 D+308 to 4.940656 D-324
Currency	1.7976931 D+308 to 4.940656 D-324
• General	
Logic	A "true" or "false" character
*Date (US)	MM-DD-YYYY (MM=month, DD=day, YYYY=year) 01-01-1900 to 11-17-2065
*Date (European)	DDD-MM-YYYY 01-01-1900 to 11-17-2065
*Phone Number	(###) ###-####
*Zip Code	#####-####
*Social Security	###-###-####
• Special	
Relational	Relates the current field to a field in another file
Mult-Choice Array	Presents a vertical menu of choices
Push Button	Returns user assigned key code when activated
Mouse Field	Returns user assigned key code when activated
Horizontal Scroll Bar	-32768 to 32767
Vertical Scroll Bar	-32768 to 32767

Fields designated with an asterisk (*) generate mask characters

Table 10: Field Types

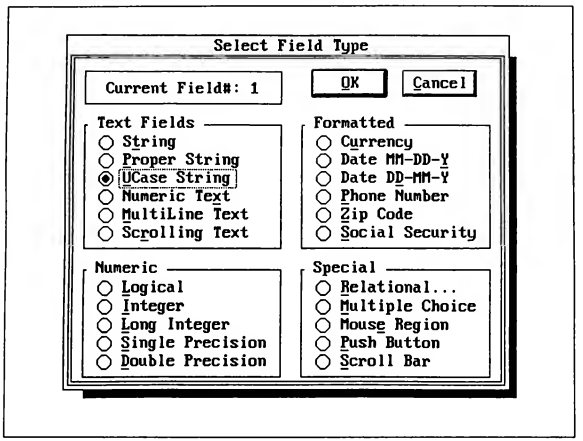


Figure 25: Field Type Dialog Box

Field Settings

Each field has a group of user-assigned attributes that collectively are called Field Settings. These attributes are set using dialog boxes, one of which is depicted in figure 26. This figure shows an example Field Settings dialog box which is generated for the Currency field type. It is important to realize that certain fields will generate dialog boxes containing slightly different options. For example, the push Button field type dialog box queries for a key code value to be returned when activated.

The following pages present an alphabetized list of input elements which are encountered for the Field Settings dialog box.

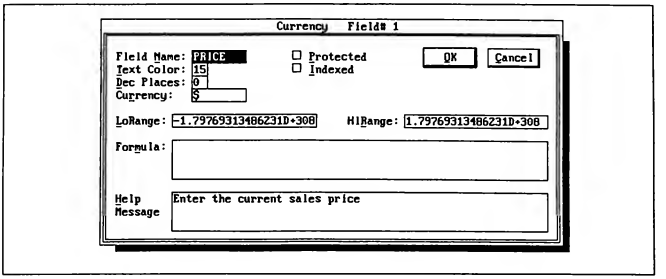


Figure 26: Currency Field Dialog Box

- **Currency Symbol**

  The symbol to be used when displaying currency values, such as $ for dollars or ¥ for Yen.

- **Decimal Places**

  Number of digits to be displayed after the decimal point. The value of the number presented will be rounded based on the internal decimal representation.

- **False Character**

  The character to be accepted as False in a logical field. The space bar will toggle True and False characters.

- **Field Name**

  A unique name, up to eight-characters in length, for the current field. This name is generated automatically but can be changed to any name you feel is appropriate. The name may be used as a variable or a string, and can appear in field formula calculations.

  Realize that a constant, function, or operator name such as ARCSIN represents a reserved word and should not be used as a field name unless it will not be used in calculated fields.

- **Formula**

  Formulas are considered to be string formulas if they are associated with non-numeric fields.

  Formulas are considered to be numeric formulas and define calculated fields if they are assigned to numeric fields. Numeric fields include: integer, long integer, single precision, double precision, US date, European date, and horizontal or vertical scroll bars.

While string formulas are limited to string concatenation, numeric formulas may use a variety of functions, constants, and operators. Please see the next section, *Numeric Formulas* for more detail.

- **Help Message**

  The text specified in the Help Message text box is the field-sensitive help which is presented in a window on the form when **F1** is pressed.

- **Highlight Color**

  The highlight color specifies the color to use when a mouse field is selected. Since the specified highlight color will be XORed over the existing color, the actual color will vary with the underlying color. A value of 0 will not display a highlight. To determine a specific highlight color over a given background color use this simple formula:

  Color = (Existing Color) XOR (Desired Highlight Color)

  For example, if the mouse region is blue and you want the highlight to appear red:

COLOR	COLOR #	COLOR # (BINARY)
Blue	1	00000001
XOR Red	XOR 4	XOR 00000100
Violet	5	00000101

  You would therefore assign color 5 (normally violet) as the highlight color in the field definition dialog box. You can use the BASIC editor's Immediate mode to calculate the appropriate value: PRINT 1 XOR 4.

- **Indexed Field**

  This input element should be checked if the current field is to be indexed.

  Indexed fields are not processed by Graphics QuickScreen; however, this information can be used as a flag by the calling program to determine which fields in a form are to be indexed. Note however that the Fld(N).Indexed variable is also used to hold the small change value for scroll bars. If you are testing to see if a field has been indexed on a form that has scroll bars, make sure that scroll bars are excluded in the search.

- **Key Code**

  The key code that is to be returned when a push button is pressed. The value should correspond to the character's ASCII code. Extended keys are defined as the negative value of the character's ASCII code without

the leading CHR$(0). Some common key codes are shown in the following example:

Key	*Key Code*
Enter	13
Esc	27
A	65
F10	-68
PgUp	-73

- **Large Change**

  When a scroll bar is active, this value indicates the amount of change when clicking on the sliding portion of the scroll bar or when pressing **PgUp** or **PgDn**.

- **No Formatting**

  This input element should be checked if a number is to be displayed as it was entered by the user. If this option remains unchecked, then numbers are right-justified.

- **Protected Field**

  This input element should be checked if the current field is to be protected against modification. That is, the field will be a display-only field. Any field type can be protected but there must be at least one non-protected field on each form.

- **Range**

  Specifies the upper and lower limits for numeric input, or the date range for date input between which entered values must fall before being accepted in a form.

- **Relational Field**

  This option lets you specify a file and field name for a relational field. A calling program may use this information to form a relational link to data in another data file.

  Relational fields are not processed by Graphics QuickScreen; however, the related file name and the related field number are available to the calling program. See the discussion of the FieldInfoG TYPE variable under *FLDINFO.BI* in the *Routines* section for further information.

- **Small Change**

  For scroll bars, this value indicates the amount of change when the cursor keys are used or the up/down scroll buttons are clicked.

- **Tab Color**

  For Mouse fields, the color to use in order to indicate the outline of the field whenever tabbed to or selected with a mouse. A value of 0 will not display an outline. Since the specified Tab color will be XORed over the existing color, the actual color will vary with the underlying color. To determine a specific outline color over a given background color use this simple formula:

```
Color = (Existing Color) XOR (Desired Tab Color)
```

  For example, if the background region is blue and you want the outline to appear red:

COLOR	*COLOR #*	*COLOR # (BINARY)*
Blue	1	00000001
XOR Red	XOR 4	XOR 00000100
Violet	5	00000101

  You would therefore assign color 5 (normally violet) as the Tab color in the field definition dialog box.

- **Text Color**

  Specifies the text foreground color for the field. The text background color will be whatever color the field was painted.

- **Toggle**

  When checked, a mouse field will toggle between the highlight color and the original color each time the field is activated. (When highlighted, Form$(N, 0) will contain an "X", otherwise it will hold a single space.)

- **True Character**

  The character to be accepted as true in a logical field. The space bar will toggle between true and false characters during operation.

- **Numeric Formulas**

  Field calculation formulas in Graphics QuickScreen use the same syntax that BASIC uses. For example, you would calculate the total price for an item which costs PURCHASE dollars (where PURCHASE is a field name) and is taxed at 6% as follows:

```
PURCHASE + (PURCHASE * .06)
```

  Notice that parentheses may be used to group parts of a formula to develop a mathematical hierarchy. Although this is a very simple example, the

formula can use a variety of functions, constants, and special operators. These are summarized below:

- Functions

Graphics QuickScreen supports many functions available in BASIC as well as several that are not. The available functions are summarized in Table 11.

Name	Value
ARCSINH	Inverse Hyperbolic Sine
ARCCOSH	Inverse Hyperbolic Cosine
ARCTANH	Inverse Hyperbolic Tangent
ARCSECH	Inverse Hyperbolic Secant
ARCCSCH	Inverse Hyperbolic Cosecant
ARCCOTH	Inverse Hyperbolic Cotangent
ARCSIN	Inverse Sine
ARCCOS	Inverse Cosine
ARCSEC	Inverse Secant
ARCCSC	Inverse Cosecant
ARCCOT	Inverse Cotangent
SINH	Hyperbolic Sine
TANH	Hyperbolic Tangent
SECH	Hyperbolic Secant
CSCH	Hyperbolic Cosecant
COTH	Hyperbolic Cotangent
CSC	Cosecant
COT	Cotangent
SEC	Secant
SIN	Sine
COS	Cosine
TAN	Tangent
ATN	Arc Tangent
LOG	Natural Log
EXP	Exponent
SQR	Square Root
CLG	Common Log
!	Factorial
ABS	Absolute value

Table 11: Graphics QuickScreen Field Functions

- Constants

Several constant names may be used in a formula expression. The field constants available are summarized in Table 12.

Name	Value
Pi	3.14159265358979323846
E	2.718281828459045

Table 12: Graphics QuickScreen Field Constants

- Math Operators

Table 13 presents the Graphics QuickScreen math operators which may be used just like BASIC's.

- Relational Operators

Relational operators, presented in Table 14, compare numerical values. When true, the formula expression evaluates to -1; when false the result is 0.

Operator	Purpose
^	Power
*	Multiplication
/	Division
\	Integer Division
MOD	Module
+	Addition
-	Subtraction

Table 13: Graphics QuickScreen Field Math Operators

Operator	Purpose
=	Equal
>	Greater than
<	Less than

Table 14 Graphics QuickScreen Field Relational Operators

- Boolean (Logical) Operators

  You may use Boolean operators in a numerical formula to evaluate expressions to True, which is -1, or False, which is 0. Table 15 summarizes Graphics QuickScreen's Boolean operators.

Boolean Operators
AND
NOT
OR

Table 15: Graphics QuickScreen Field Boolean Operators

Scroll Bars

Scroll bars allow the user to enter a number from a range of values. Upper and lower limits can be set to any values ranging from -32768 to 32767. Numbers can be returned in any step increment that you specify.

Two different step sizes are available for each scroll bar. These are specified in the field setting dialog box in the Small Change and Large Change text boxes. The Small Change value specifies the amount of change when using the **Up/Down/Left/Right** cursor arrow keys or when clicking on the direction scroll buttons. The Large Change value specifies the amount of change when clicking on the sliding portion of the scroll bar or when pressing **PgUp** or **PgDn**. In addition, pressing the **Home** key selects the low limit while the **End** key selects the high limit.

The lower limit must always be less than the upper limit. To make the scroll bars read backwards such that the high value is indicated when the scroll pointer is at the top or left of a scroll bar, subtract Fld(N).Value from Fld(N).HiRange. See *Handling Scroll Bars* for more information.

5

Creating Screens

Creating
Screens

CREATING SCREENS

Graphics QuickScreen's editing features make it very easy to create screens and embellish them with sophisticated graphics and color. Its user interface allows you to paint your data entry screens much as you would paint screens in a conventional paint program.

Because Graphics QuickScreen screens are saved in the popular .PCX format, you may also import scanned .PCX images or use screens created in any other paint program that uses the .PCX format. The imported screens must also have been saved in either the 640x350 or 640x480 16-color mode. These screens can then be loaded into the Graphics QuickScreen editor for further enhancements and the addition of data entry fields.

It is important to understand that the graphic image of the forms you create and the field definitions themselves are stored in two completely separate and independent files. The screens that you paint are simply blank *forms* on which to perform data entry. What you draw affects only what background color is used for text fields and identifies the placement and color for push buttons and scroll bars.

The screen image can be displayed with or without field definitions—data entry can be performed with or without the original screen—it simply uses whatever colors are present for text background colors and uses the previously defined foreground color for the actual text. Of course, it is intended that you display the correct screen along with its related form definition.

The process for creating data entry screens in the Graphics QuickScreen editor can be broken down into four basic steps:

1. Design and draw/paint the blank form
2. Define the fields
3. Test the form
4. Generate BASIC source code

Graphics QuickScreen's interface allows you to perform these steps in almost any order, but there are several points to consider.

When designing your screens it is helpful to indicate the boundaries of data entry fields by painting them a specific color. This helps you to identify field boundaries when defining fields within the Graphics QuickScreen editor but more importantly indicates to your users where fields

begin and end. When your program runs, the EditFormG subroutine reads the existing background color from the screen for each field as it is encountered. EditFormG uses whatever background color it finds whenever printing text to the field.

Since all text entry fields must be located at standard text rows and columns, you must be careful when painting fields to locate the field's coordinates correctly. This is easily accomplished by setting Grid Snap values that correspond to the current screen mode's text size. (See *Painting Fields* for more information.) Mouse fields, push buttons, scroll bars, and constant text are not restricted by text rows and columns and can be placed at any pixel location.

Creating Data Entry Forms

6

Creating Data Entry Forms

CREATING DATA ENTRY FORMS

Defining Fields

The following sequence shows the steps needed to define any data field:

1. Choose the **Define Data Fields** command
2. Position the cursor
3. Choose a field type
4. Adjust the field size
5. Complete field settings

To create a field, first choose the **(Compose-Fields) Define Data Fields** menu option. Second, move the cursor to the starting position of the field you are defining, and then press **Enter** or double click the mouse. If you are defining mouse fields, push buttons, or scroll bars, the initial starting point is irrelevant since it will be redefined after the next step.

Third, choose a field type from those presented. Fourth, adjust the field size, keeping in mind that you must allow enough space to hold the field's data. You should consider that dashes, commas, and other "mask" characters occupy space in the field and should be considered when adjusting the size.

If you are using a form and notice that a numeric field is filled with percent (%) symbols, then the data in that field is exceeding the field's size. This usually indicates that the field must be made larger.

If you selected mouse field, push button or scroll bar, you will be prompted to define the field by drawing a box around it. For push buttons and scroll bars to function properly, this box must overlay exactly the black outline of the image. When the size is correct, all sides of the encompassing box will appear yellow.

The last step is to complete the field settings by specifying the field name, text foreground color (if applicable), associated formula, help message, and other available options. Field names are assigned automatically by Graphics QuickScreen, but they may be changed to any other unique name.

The field definition procedure outlined becomes cyclic: step 5 is followed by step 2. This cycle continues until either **Esc** or **F10** is pressed after step 5. The final step 5 presents a dialog box which is appropriate for the field being defined. This dialog box, therefore, is not always the same.

Try to define the fields in the same order you wish them to be used in the form. This reduces the likelihood that you will need to use the field Rearrange features, discussed later in this section.

Often, you will notice ways that the form can be improved once you create it. Editing fields already on the screen is easy. Simply access the **(Compose-Fields) Define Data Fields** menu option. You can use the + and - keys to access the next and previous fields, respectively, in the form. Pressing **Del** deletes a field from the form, while pressing **Ins** inserts a field before the current field.

You can also directly access any predefined field during step 2 by pressing the **Tab** key, or by clicking on the field's number shown in the dialog box. The field's number will become a text box allowing you to enter the desired field number. Press **Enter** or click the mouse anywhere outside of the field number's text box to jump to the specified field. Entering a number greater than or equal to the highest field number will select the last field.

Rearranging Fields

If you need to rearrange the order of fields on a form you will use the **(Compose-Fields) Rearrange Data Fields...** command. This generates the dialog box similar to the one shown in Figure 27.

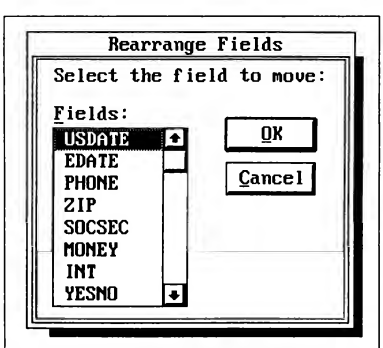


Figure 27: Rearrange Fields Dialog Box

Fields may be re-ordered in the **Rearrange** dialog box by first selecting a field in the list box and then clicking the OK push button. You will then

be able to position the selected field above or below another field in the list box. Clicking OK again inserts the selected field at the indicated position. Clicking Cancel will instead exit the dialog box.

Printing Field Definitions

The field definitions for the current screen can be printed* by using the **(Compose-Fields) Print Field Definitions** command. If the printer is ready then the fields will be printed. The heading of the report will contain the field file name, the record length, and the current date and time. Page numbers will appear at the upper-right corner for your convenience. The default printer port is LPT1, though this may be changed from the **System** dialog box under the **Settings** menu.

The remainder of the report consists of a columnar table containing the headings summarized in Table 16.

Heading Name	Meaning
Fld	The field number
Offset	The integer pointer representing a byte offset of this field in the entire field structure
Name	The field name up to eight characters
Type	The field type; See Table 20, FieldInfoG FType constants. If a push button field, the key code to be returned is also displayed
FldLen	For text fields, the field length in bytes; for mouse fields, push buttons, and scroll bars, the height and width in pixels of the field
RecLen	The record length in bytes
Located	For text fields, the row and column coordinates of the field; For mouse fields, push buttons and scroll bars, the X/Y coordinates in pixels of the upper left corner of the field
Related File	The file name for relation
Index	Yes or No; tells whether the field is to be indexed
Prot	Yes or No; tells whether the field is protected
Range	For numeric fields, the upper and lower range for allowable input; for scroll bars, the maximum and minimum values to be returned
Formula	The defined field formula

Table 16: Field Report Headings

* The LPT? number is specified in the **System** dialog box under the Settings menu. The default is for LPT1.

Saving A Form

Creating a data entry screen usually takes both care and time. You will want to save often while designing or making changes to your form by using the **(File) Save...** menu command.

It is important to understand that Graphics QuickScreen provides a safeguard to protect you from destroying an existing form (.FRM) file inadvertently. If you load a form and make changes to it, Graphics QuickScreen first checks to see if a data file exists which has the same name as the form. For instance, if you are editing MYFORM.FRM and the file MYFORM.DBF or MYFORM.DAT exists, then Graphics QuickScreen will not overwrite the existing .FRM file (since doing so could make the existing data file unreadable). Instead, a file with the extension of .NEW is created. In this example, MYFORM.NEW would be created instead of MYFORM.FRM.

Files created by Graphics QuickScreen

When you save screens created with the Graphics QuickScreen editor, several files are created. The screen image is saved in compressed form in the .PCX format. Information describing the fields you have defined will be saved in a separate .FRM file, and can optionally be saved as a BASIC source file. A .BI (Basic Include) file containing a TYPE definition that corresponds to the field types in your form is also created.

To display and edit forms in your program, you will first need to assign the form definitions arrays from either the .FRM file or the .BAS source file. The GetFldDefG BASIC subroutine is provided to read the .FRM file and assign it to two arrays, Fld() and Form$().

If you prefer to code the field definitions directly into your source code, you can instead call the BASIC subroutine optionally created when you saved your form. This source file will have the same name as your form's .PCX file, but with a .BAS extension. When called, it performs the same function as the GetFldDefG subroutine.

The Fld() array is a TYPE array that contains field attributes such as the field's row, left column, right column, field type, and so on. See the FLDINFO.BI TYPE structure definition for more information.

The Form$() array is a three-dimensional string array that contains field formulas, help messages, and also holds the actual data that is assigned to or entered into each field. The information contained in element 1 of each of the arrays relates to field 1 on your form, element 2 relates to field 2, and so forth.

Element 0 of both the Fld() TYPE array and the Form$() array contain general form information. The Fld(0).xxx elements contain information such as the total number of fields, total length of the form, and the number of text rows. Form$(0, 0) is of particular interest because it contains all field input combined into a single string ready to be written to disk. Numbers are placed into Form$(0, 0) as IEEE formatted strings. This string can then be copied directly into a variable of the TYPE structure defined in the .BI file created when you saved your form. The entire form can then be saved as a record in a random file by PUTing this TYPE variable to disk.

After the field definitions have been assigned, forms are displayed by calling the ShowForm subroutine. This routine first sets the proper screen mode, color palette, and number of screen rows, and then displays the screen.

Once the form definitions have been loaded and the screen displayed, data is entered into the form by calling the EditFormG subroutine. EditFormG is a pollable routine that reads the field information and handles all user input while editing the form. It also returns several variables that may be examined to determine the current editing status.

Since EditFormG is meant to be called in a loop, the calling program can monitor all editing as it occurs. This gives the calling program the ability to test for specific keys or conditions and act upon them independently, instead of having to wait for the user to press **Enter** or **Escape**. The current field number, last key pressed, insert status, and the current mouse X and Y coordinates are just some of the parameters that can be monitored at any given moment.

All data entered into your form is stored as a string in the Form$(N, 0) array. (N represents the field number.) Numeric fields are stored literally as they appear on the screen. Field data is also stored in Form$(0, 0) as a single continuous string that has numbers and dates formatted as IEEE formatted strings.

When editing is complete, all user input in the form is contained in the Form$() array. If you wish to save your form as a record in a random file, the SaveRec and GetRec routines are provided to save and retrieve this data.

7

Routines

Routines

GRAPHICS QUICKSCREEN ROUTINES

Procedure Reference Section

Graphics QuickScreen allows the programmer to generate screens and process forms from BASIC using a variety of options. This flexibility necessarily brings some complexity. In order to make this section most useful, we first present some terms and concepts with which you should be familiar. Then we introduce the important Include files which you will use in your calling programs. Next, we examine the variables which play a vital role in using Graphics QuickScreen from BASIC and which appear in the demonstration programs. Finally, we present documentation for the Graphics QuickScreen BASIC and assembler routines you will be using.

Integers

Throughout the remainder of this manual we will make reference to several important integer variables, such as Action and ErrorCode. As you may know, such variables are represented in BASIC with a trailing percent sign (%). Thus, X% refers to an integer variable. You will notice, however, that many examples and discussions which use integer variables omit the percent symbol. The reason is that our sample programs and program fragments assume the presence of a DEFINT A-Z statement, which ensures that variables lacking a type identifier are integers by default.

We have retained the type identifier when showing the calling syntax for the Graphics QuickScreen routines to clearly show which parameters are integers.

Parameters

A parameter is a variable which appears at the top of a subprogram or function heading. For example:

```
SUB GPrintOVE(Row%, Col%, Text$, Colr%)
```

Here, Row%, Col%, Text$ and Colr% are variables in the parameter list. There must always be a one-to-one correspondence in both number and type for the arguments and the parameters used when implementing a routine. For instance, if a particular routine was designed to accept 5 parameters, then you must pass exactly 5 arguments to the routine when you call it. And if the first parameter expects an integer, you must use an integer as the first argument.

Arguments

An argument is a variable or value used when calling a subprogram or invoking a function. For example:

```
CALL GPrintOVE(Row%, Col%, Text$, Colr%)
```

Here, Row%, Col%, Text$ and Colr% are the argument list. The variable Text$ could be replaced by the literal "Hi there!" if desired. Similarly, the integer argument Colr% could be replaced with the integer constant 12. Arguments are passed to and used by the subroutine being called. When arguments are variable names (rather than numbers or strings) the subroutine being called may modify them and make their new values available to the calling program.

Action

The action parameter is used by many of Crescent Software's pollable routines. The integer value contained in Action tells the called subprogram what it should do. Graphics QuickScreen uses a pollable routine called EditFormG which serves as the core forms-processing subroutine. When using EditFormG, Action may be set to 1, -1, -2 or 3. Table 17 summarizes how these values affect the operation of EditFormG.

Action Value	Meaning to EditFormG
1	Initializes the current form for editing; pads all Form$() elements to their proper lengths and formats; displays the contents of all fields in the form; resets Action to 3
-1	Same as Action 1 except it resets the push button and mouse field status
-2	Restores a push button to its non-active state
3	Keeps polling the current form while editing continues

Table 17: Action Values for EditFormG

Note that Action -1 is used only when a push button or highlighted mouse field is used to call up another form. This prevents the push button or mouse field from the first form from being restored (displayed) on the second form.

Action -2 is used only to restore a pushbutton to its non-active state after it was used to display another partial form over the existing one.

Form$() Array

The Form$() array is a conventional (not fixed-length) 3-dimensional string array used to store information both about a form and about the fields it contains. The first subscript must be dimensioned to the total number of fields in the form; the second subscript is always dimensioned to 2.

A special area of the Form$() array called the *form buffer* is stored in the array element Form$(0, 0). The form buffer collects information from all fields and formats them into a single fixed-length structure. This lets you use random file commands to quickly load and save form information.

All of the fields in the form are stored in the Form$(0, 0) array element with one exception: data from notes fields are stored in separate notes files having a .NOT extension. (A multi-line text field is considered a note field.) When a notes field is in the form buffer, four bytes are reserved to hold an offset into the .NOT notes file. The position in the .NOT file pointed to contains a two-byte integer value which gives the length of the note. In this way a linked list is created between the form buffer and the current notes file.

Form$(N, 0) contains the value of field N; Form$(N, 1) contains the help message for field N; and Form$(N, 2) contains the formula for field N. Note that some fields will not have a help message or formula, and, in such cases, Form$(N, 1) and Form$(N, 2) would be null.

The organization of the Form$() string array is summarized in Table 18.

Form$()	Element	Description
Form$	(0, 0)	Holds all data from fields as a contiguous string with numbers as IEEE formatted strings
Form$	(FieldNo, 0)	Holds data (numbers are stored as formatted strings)
Form$	(FieldNo, 1)	Holds help message string
Form$	(FieldNo, 2)	Holds formula for calculated fields

Table 18: Form$() Layout

Type Variables And Constants

Recall that Include files are ASCII text files containing BASIC source code. In general, they contain source code which is used by more than one program. Placing such code in external files makes it easy to include them in a program without having to retype their contents each time. Also, if changes are made to an Include file, all of the programs that reference it will be updated the next time they are compiled. The $INCLUDE metacommand may be inserted anywhere in a program using this syntax:

```
'$INCLUDE: '[d:][\Path\]FILENAME.EXT'
```

When this line is encountered by the compiler, the contents of the path and file name enclosed in single quotes are read and compiled. If the file cannot be found on the directory specified, then the path stored in the INCLUDE environment variable is accessed. If the file still cannot be located, then the compiler reports an error.

There are three important Graphics QuickScreen-related Include files:

SETUP.BAS
FLDINFO.BI
EDITFORM.BI

The .BI extension is a Microsoft conventions and stands for "**B**ASIC **I**nclude".

SETUP.BAS

SETUP.BAS is an include file that should be entered as the first executable statement in your programs. It determines the current monitor type and sets up the GPDat%() array. The GPDat%() array is a COMMON SHARED array that contains useful information about the current form and screen mode. This information is used by several of the Graphics QuickScreen subroutines. See Appendix A for a complete description of the GPDat%() array.

FLDINFO.BI

FLDINFO.BI includes the field information TYPE array and several constant assignments. This user-defined TYPE is required by a calling program to obtain information about a field in the currently-active form. The FldInfo TYPE structure looks like this:

```
TYPE FieldInfoG
  Fields     AS INTEGER
  Row        AS INTEGER
```

```
    LCol       AS INTEGER
    RCol       AS INTEGER
    StorLen    AS INTEGER
    FType      AS INTEGER
    RelFile    AS STRING * 8
    RelFld     AS INTEGER
    Indexed    AS INTEGER
    FldName    AS STRING * 8
    Decimals   AS INTEGER
    RelHandle AS INTEGER
    Protected AS INTEGER
    ScratchI   AS INTEGER
    LowRange   AS DOUBLE
    HiRange    AS DOUBLE
    ScratchS   AS STRING * 8
    Value      AS INTEGER
END TYPE
```

As in the demonstration programs, you must create and dimension the Fld() TYPE array as follows so that it is defined as the FieldInfoG TYPE:

```
REDIM Fld(0) AS FieldInfoG
```

Dimensioning any array to zero elements simply defines the array while committing the smallest block of memory possible. The Fld() array will be redimensioned later to the actual number of fields in the current form using the NumFieldsG function or by calling the optional form definition BASIC module. This way the array will be made only as large as it needs to be.

When a form is loaded, the calling program may obtain specific information about each field using the Fld() TYPE array. For example, to find out whether the field number 3 is protected the calling program would use a statement like:

```
IF Fld(3).Protected THEN ...
```

The program can also access a field's position on the screen and for applicable fields, its low and high ranges for acceptable data entry. As you can see by examining the TYPE definition, many other field characteristics are available to your program as well.

In addition to examining the contents of the Fld() TYPE array, a calling program may also change these values. This means that a field can be protected or unprotected at runtime as the form is being processed. Or, based on values entered somewhere else on the form, low and high ranges for certain fields can be adjusted.

The Fld() TYPE array reserves element 0 for special use. For example, Fld(0).StorLen contains the record length, in bytes, of the entire current

form. However, Fld(1).Row contains the row position for field 1 on the form. Only Fld().Fields, Fld().Row, Fld().LCol, Fld().StorLen, Fld().Value and Fld().Indexed make use of the zero element, however.

Table 19 summarizes the FieldInfoG TYPE elements. When "N" is mentioned, it applies to the subscript in the Frm() array: for Frm(0), N is equal to 0.

FieldInfoG's FType element constants are summarized in Table 20.

Element	Description
Fields	When N = 0, the number of fields in the form; when N > 0, the field's integer offset position within the Form$(0, 0) element
Row	When N = 0, the upper row of a partial .PCX screen; when N > 0, the screen row position for field N; for mouse fields, push buttons, or scroll bars, the upper left row of the field in pixels
LCol	When N = 0, the left column of a partial .PCX screen; when N > 0, the screen left column of the field; for mouse fields, push buttons and scroll bars, the upper left column of the field in pixels
RCol	When N = 0, the width in columns of a partial .PCX screen; when N > 0, the right column of the field. For mouse fields, push buttons and scroll bars, RCol returns the right column of the field in pixels
StorLen	When N = 0 the record length of the form; when N > 0, the number of bytes required to store the contents of field N on disk
FType	The field type number (see table 20)
RelFile	If a relational field, the base name of the file for relation
RelFld	If a relational field, the number of the relational field; for scroll bars, this component stores the large change value

(Continued ...)

Indexed	0 when field is not indexed; -1 if field is indexed; for scroll bars, this component stores the small change value
FldName	The name of the current field
Decimals	When N = 0, the height in pixels of a partial .PCX screen; when N>0, the number of decimal places used for numeric fields; if -1, numbers are not formatted; for Scrolling text fields, the text input filter (1-5) is set or returned
RelHandle	The BASIC file number for the related file; this number may be used for GET # and PRINT # statements; for scroll bars, the current screen pointer position
Protected	0 if field is protected, -1 if it is not
ScratchI	For mouse fields, push buttons and scroll bars, the bottom right corner row of the field; for Scrolling Text fields, the character to display at the left of the edit window; for Multi Line text fields, the bottom screen row of the field; unused for all other field types
LowRange	For numeric fields, the low range limit, for scroll bars, the minimum value to be returned; for toggling mouse fields, LowRange holds a value of -1 or -2. -2 indicates that the field is currently highlighted.
HiRange	For numeric fields, the high range limit; for scroll bars, the maximum value to be returned
ScratchS	The currency type character for Currency fields. It can be used by you for other fields to store miscellaneous information such as flags, etc.
Value	For text fields, the text foreground color; for mouse fields and push buttons, the keycode that is to be returned; for scroll bars, the current scroll bar value

Table 19: FieldInfoG TYPE elements

The constant assignments in the FLDINFO.BI file make it easy to use the FTYPE element of the FieldInfoG TYPE. For example, if you need to know if the current field is a Proper String, you could use a statement similar to:

```
IF Fld(CurField).FType = PropStrFld THEN ...
```

CONSTANT	DECLARATION
CONST StrFld = 1	String
CONST PropStrFld = 2	Proper string
CONST UCaseStrFld = 3	Upper case string
CONST NumericStrFld = 4	Numeric string
CONST NotesFld = 5	Notes (multi-line text)
CONST ScrollFld = 6	Scrolling text (single-line)
CONST LogicalFld = 7	Logical
CONST IntFld = 8	Integer
CONST LongIntFld = 9	Long integer
CONST SngFld = 10	Single precision
CONST DblFld = 11	Double precision
CONST MoneyFld = 12	Currency
CONST DateFld = 13	US date
CONST EuroDateFld = 14	European date
CONST PhoneFld = 15	Telephone number
CONST ZipFld = 16	Zip code
CONST SoSecFld = 17	Social security number
CONST Relational = 18	Relational
CONST MultChAFld = 19	Multiple choice array
CONST MouseFld = 20	Mouse field
CONST PButton = 21	Push button
CONST HScrollFld = 22	Horizontal scroll bar
CONST VScrollFld = 23	Vertical scroll bar

Table 20: FieldInfoG FType Constants

EDITFORM.BI

EDITFORM.BI contains constant assignments and the user-defined TYPE FormInfoG which is constructed as follows:

```
TYPE FormInfoG
   StartEl   AS INTEGER
   FldNo     AS INTEGER
   PrevFld   AS INTEGER
   FldEdited AS INTEGER
   KeyCode   AS INTEGER
   TxtPos    AS INTEGER
   InsStat   AS INTEGER
   Presses   AS INTEGER
   MRow      AS INTEGER
```

```
    MCol       AS INTEGER
    Button     AS INTEGER
    Mx         AS INTEGER
    My         AS INTEGER
    DoingMult  AS INTEGER
    Edited     AS INTEGER
END TYPE
```

The FormInfoG TYPE elements are explained in Table 21. You will use the following statement to create the Frm TYPE variable:

```
DIM Frm as FormInfoG
```

The Frm TYPE variable is used by a calling program to set the current field to be edited, to examine the last key pressed, to toggle the insert status of the forms editor and to determine when data in a form has been altered.

This last item, Edited, because it lets you know if any of the information on the form has been changed. Each time you update a record in the file you should set Frm.Edited to 0. Then, if any field values are changed, Frm.Edited will be set to -1, letting you know that it is necessary to write the form record to the file again.

You may read or set any of the Frm TYPE elements in your program. However the elements Presses, Mx, My, MRow, and MCol should only be read—altering them will have no affect on the form.

FormInfoG Element	Description
StartEl	Starting (base) element of the current form.
FldNo	Current field number
PrevFld	Previous field number (different from FldNo only when first moving to a new field
FldEdited	Set to -1 if a field has been changed
KeyCode	ASCII value of the last key pressed; extended keys return a negative value (for example, **F1** = -59)
TxtPos	Cursor position relative to current field
InsStat	Current insertion mode status (-1 = insert status is On)
Presses	Number of mouse presses since last press
MRow	Mouse row number at last press
MCol	Mouse column number at last press
Button	Current mouse button
Mx	Current mouse X position
My	Current mouse Y position
DoingMult	Set to -1 if handling a multiple choice field

Edited	Set to -1 if any field on the form has changed by the user

Table 21: FormInfoG TYPE Elements

To summarize, there are several important variables. The GPDat%() array contains system environment and color information. The Fld() TYPE array provides information for each field in a form. The Frm TYPE variable gives information about the form itself and about user-oriented events.

GRAPHICS QUICKSCREEN ROUTINES

Several BASIC modules are supplied with Graphics QuickScreen that you will call from your programs. The listing which follows shows the supplied module (file) names and the subroutines they contain. Note that module names are shown in uppercase, while subprogram names are given as mixed-case.

EVALUATE.BAS: Expression evaluation routine for calculated fields

Evaluate	Returns the value of an expression

EDITFORM.BAS: For handling data entry on a form

EditFormG	The main routine for data entry
EndOfForms	Returns last field on a form/forms (for multi-page)
FixDate	Turns dates such as "2-3-1991" into "02-03-1990"
FldNum	Returns a field number given a field name
Format	Places a formatted version of a number into the form
Message	Used to display/clear a message box
PrintArray	Displays the contents of the Form$() array on form
PressPButton	Simulates pushing a push button
ReleasePButton	Simulates releasing a push button
SaveField	Formats data and saves it to the Form$(0, 0) buffer
ShadowBox	Draws a 3D rectangle (used with Message)

UnPackBuffer	Copies data from Form$(0, 0) into individual elements
Value	Returns value of a formatted numeric string like "$1,200.00"

FRMFILE.BAS: For loading .FRM form definition files

GetFldDefG	Loads an .FRM file into the supplied arrays
NumFieldsG	Determines the number of fields contained in a .FRM file

GQEDITS.BAS: Multi-line edit routine

QEdit	Multi-line edit routine used for Notes fields

GQSCALC.BAS: Support routines for performing field calculations.

CalcFields	Recalculates dependent fields based on a given field
Tokenize	Resolves field name references in a formula to their field numbers
WholeWordIn	Searches a string for a Whole Word version of a sub-string

GDISPLAY.BAS: Support routines for displaying screens

ShowForm	Main routine for displaying screens
MoveFrm	Repositions field coordinates for partial screens

LIBFILE.BAS:	For loading .PCX, .GMP, and .FRM files from a custom .GSL library.
FindLibFile	Returns the specified file's size and offset within the .GSL library
LibGetFldDefG	Loads an .FRM file contained in a custom .GSL library into the supplied arrays

LibGetGMP	Loads the specified .GMP file from a custom .GSL library into an array.
LibNumFieldsG	Determines the number of fields in an .FRM file contained in a custom .GSL library
LibShowForm	Displays the specified .PCX screen from a custom .GSL library

LISTBOX.BAS: Vertical menu routine used for multiple choice fields

ListBox	Menu subroutine used for multiple-choice fields

Not all modules or subprograms and functions will be useful to you. However, a few in particular are required for certain tasks. Table 22 lists which modules and calls are required when working with certain types of Graphics QuickScreen files.

Method	Module(s)	You Call
Display Screens (.PCX)	GDISPLAY.BAS	ShowForm
(.PCX in .GSL Library)	LIBFILE.BAS	LibShowForm
Screen Files (.GMP)	GETGMP.BAS	GetGMP
(.GMP in .GSL library)	LIBFILE.BAS	LibGetGMP
Form Files (.FRM)	FRMFILE.BAS	NumFieldsG GetFldDefG
(.FRM in .GSL library)	LIBFILE.BAS	LibNumFieldSG LibGetFldDefG
Perform Data Entry	EDITFORM.BAS	EditFormG
Form Files (.BAS)	*MYFORM*.BAS EDITFORM.BAS	*MyForm* EditFormG

Table 22: Ways to Display Screens and Handle Forms

Table 23 lists the files needed for certain key Graphics QuickScreen features as well as the equivalent files that remove those features when they are not needed. Please understand that one of these files is needed to successfully link your program. For example, if you need the calculated fields feature in a program that you are writing, then you must compile the GQSCALCG.BAS and link that with your main program. Otherwise, you should compile the NOCALCG.BAS "stub" file and link that with your program instead.

Note that no harm is done if you use the full-featured version of a file, even when a particular feature is not needed. However, your final .EXE program will be larger than necessary because code that is not needed is added to it.

Calculated Fields:
GQSCALC.BAS (For support)
NOCALCG.BAS (To remove support)
Multiple Choice Fields:
LISTBOX.BAS (For support)
NOMULTG.BAS (To remove support)
Multi-Line Notes Fields:
GQEDITS.BAS (For support)
NONOTES.BAS (To remove support)
Scroll Bars:
SCROLLB.BAS (For support)
NOSCROLB.BAS (To remove support)
Scrolling Text Fields:
SCROLLIN.BAS (For support)
NOSCRO'.L.BAS (To remove support)

Table 23: Graphics QuickScreen optional modules

There are a number of BASIC and assembler subroutines that you may call from your programs. The following pages present an alphabetic summary of these routines. Each routine is discussed separately and we have provided information about its program type (subroutine or function), purpose, and calling syntax.

Following the calling syntax is a brief explanation of the routine's arguments. Then, a detailed discussion of the routine and each argument is presented. Finally, we have concluded each routine with either an example program segment or a reference to an example.

Some routines which are used by Graphics QuickScreen internally have been documented here so that they may be called directly from your own programs. These routines are not necessary for you to display or manage Graphics QuickScreen screens, but they may be useful to you in some other capacity.

BCopy

assembler subroutine contained in GFORMS.LIB

- Purpose

  BCopy copies of a block of memory (up to 64K in size) to a new location. It is used primarily to copy information from Form$(0, 0) to a TYPE structure.

- Syntax

```
CALL BCopy(FromSeg%, FromAddr%, ToSeg%, ToAddr%, _
  NumBytes%, Direction%)
```

- Where

  FromSeg%: Segment of the source location of the block

  FromAddr%: Address of the source location of the block

  ToSeg%: Segment of the destination

  ToAddr%: Address of the destination

  NumBytes%: Number of bytes to be copied

  Direction%: Specifies direction of the copy (0 is forward; -1 is reverse)

- Comments

  BCopy is useful in a variety of situations, such as when copying an array or duplicating a range of elements. When using the routine with forms, you will find it helpful when working with random access file I/O. As you know, you can create a TYPE structure for your form when saving forms from the Screen Designer. For instance, in the supplied CUSTOMG.FRM file, the Customer TYPE is saved to CUSTOMG.BI, and looks like this:

```
TYPE CUSTOMG
  IDNO    AS INTEGER
  DATEIN  AS INTEGER
  NAME    AS STRING * 32
  COMPANY AS STRING * 32
  ADDR1   AS STRING * 32
  ADDR2   AS STRING * 32
  CITY    AS STRING * 20
  STATE   AS STRING * 2
  ZIPCODE AS STRING * 10
  WPHONE  AS STRING * 14
  HPHONE  AS STRING * 14
  NOTES   AS LONG
END TYPE
```

CalcFields

BASIC subroutine contained in GQSCALC.BAS

- Purpose

  CalcFields is used to recalculate a field which is dependent upon other field values.

- Syntax

```
CALL CalcFields(StartOfForm%, FldNo%, Form$(),_
  Fld() AS FieldInfoG)
```

- Where

  StartOfForm%: Start of the form, equal to 0 for single-page forms; for multi-page forms, this number is equal to the offset in the Fld() TYPE array needed to point to first field of the desired form

  FldNo%: Number of the field you wish to recalculate

  Form$(): Form string array (See the section *Form$() Array*)

  Fld(): Field information TYPE array (see *FLDINFO.BI*)

- Comments

  CalcFields should be used when information related to a calculated field is changed. CalcFields looks at the value of the specified field (contained in FldNo) and recalculates all other fields which depend on it.

  CalcFields is useful only when you need to recalculate specific fields in a form.

- Example

  This example recalculates all fields that depend on the fifth field in the current form:

```
CALL CalcFields(0, 5, Form$(), Fld AS FieldInfoG)
```

Date2Num

assembler function contained in GFORMS.LIB

- Purpose

  Date2Num converts a date in string form to an equivalent integer variable.

- Syntax

  ```
  Days% = Date2Num%(D$)
  ```

- Where:

  Days%: The number of days before or after 12/31/79, and D$ is a date in the form "MMDDYY" or "MM-DD-YY" or "MM/DD/YYYY", or any similar combination

- Comments

  Because Date2Num has been defined as a function, it must be declared before it may be used.

  Date2Num is a very powerful routine with two important uses. Besides allowing what would otherwise be an eight-character string to be packed to only two bytes, it also provides an easy way to perform date arithmetic.

  Date2Num will operate on any date that is within the range 01-01-1900 to 11-17-2065. Invalid dates that fall outside of that range will return -32768 to indicate an error.

- Example

  Once a date has been converted to the equivalent integer value, you may add or subtract a number of days, and then use the companion function Num2Date to convert the result. The example below shows this in context.

```
DEFINT A-Z
DECLARE FUNCTION Date2Num(X$)
DECLARE FUNCTION Num2Date(Dat)

D$ = "09-17-88"
Start = Date2Num(D$)
Later = Start + 30
After30 = Num2Date$(Later)
PRINT "Thirty days after "; D$; " is "; After30
```

  Because Date2Num and Num2Date are set up as functions they may also be used within a print statement directly, along with optional calculations:

```
PRINT "30 days after "; D$; " is "; Num2Date$_
   (Start + 30)
```

Date2Num and Num2Date are also useful for verifying if a given date is valid, which eliminates tedious calculations that you would have to perform to take possible leap years into consideration.

The only requirement for the date validation example below is that the original date must be in the form MM-DD-YYYY, because this is the format returned by Num2Date.

```
DEFINT A-Z
DECLARE FUNCTION Date2Num%(X$)
DECLARE FUNCTION Num2Date%(Dat)

INPUT "Enter a date in the form MM-DD-YYYY: "; D$
Dat = Date2Num%(D$)
IF Num2Date$(Dat) = D$ THEN
   PRINT D$; " is a good date!"
ELSE
   PRINT "Please try again."
END IF
```

This program asks for an original date and then converts it to an equivalent number. If after converting it back to a string the result is the same, then the date that was entered is valid.

Understand that while days before 12-31-1979 are returned by Date2Num as negative values, adding and subtracting will still be performed correctly.

Please see also the companion functions Num2Date and FixDate.

DispPCXVE

assembler subroutine contained in GFORMS.LIB

- Purpose

  DispPCXVE continues the loading process started by OpenPCXFile% and displays the image to a VGA or EGA specified video page.

- Syntax

```
CALL DispPCXVE (BYVAL VideoPage%)
```

- Where

  VideoPage%: 0 for the default first display page (Visual Display Page); a value of 1 specifies the second display page

- Comments

  The parameter for this routine is passed by value to provide the maximum speed.

  The function OpenPCXFile% must be called first, as it opens the PCX file and loads the header information to determine which screen mode the PCX file is intended for.

  The DispPCXVE routine works equally well when BASIC is operating in SCREEN 7, 8, 9, 11 or 12 since the video memory for all of these modes is identical. This routine works for all the screen modes which utilize the plane system created for EGA and VGA graphics. The EGA and VGA 2-color graphics modes do not utilize the plane scheme most of the other EGA and VGA screens use. The DispPCXVE routine will still work equally well on those. Also, note that PaintBrush for Windows version 2 saves only three of the four graphics planes in the file (leaving out the intensity plane). This routine will load those files properly as well.

- Example

  See ShowForm for an example of how to use DispPCXVE.

EditFormG

BASIC subroutine contained in EDITFORM.BAS

- Purpose

  EditFormG is the core data entry routine that handles all user input, cursor, and mouse activity when processing forms. This routine is pollable so the calling routine can monitor user input as it occurs.

- Syntax

```
CALL EditFormG(Form$(), Fld(), Frm, Action%)
```

- Where

  Form$(): Form string array (see *Form$() array*)

  Fld(): Field information TYPE array (see *FLDINFO.BI*)

  Frm: Form information TYPE variable (see *EDITFORM.BI*)

  Action%: A flag used to control how the form behaves when called (see *Action*)

- Comments

  The EditFormG subprogram is a major routine in Graphics QuickScreen. It will allow calling programs to process forms, making additional programming virtually unnecessary.

  When EditFormG is called, it uses information in Form$(), Fld(), and Frm. Form$() is a conventional (not fixed-length) two-dimensional string array. The first subscript must be dimensioned to the total number of fields in the form. The second subscript must be dimensioned to 2. Fld() is a TYPE array which is dimensioned using the FieldInfoG TYPE definition. Both Form$() and Fld() may be dimensioned using the NumFieldsG function as shown below.

```
Size% = NumFieldsG%(FormName$)
REDIM Form$(Size%, 2)
REDIM Fld(Size%) AS FieldInfoG
```

  Once the form arrays are properly sized, they can be initialized and loaded using the GetFldDefG routine:

```
CALL GetFldDefG(FrmName$, StartEl%, Fld(), Form$())
```

  To continue with the list, Frm is a TYPE variable which is DIMed to the FormInfoG user-defined TYPE (please see *EDITFORM.BI*).

Action is either -2, -1, 1 or 3, as discussed earlier (please see *Action*).

- Example

The most effective way to poll EditFormG is to wait for a particular keypress, such as **Esc**, to occur. When this happens, the form may be cleared from the screen and processing may continue.

In the example below we present a DO loop showing how to poll EditFormG. The DO loop is terminated when **Esc** is pressed.

```
.
.
.
ACTION = 1
DO
  CALL EditFormG(Form$(), Fld(), Frm, Action)
LOOP UNTIL Frm.KeyCode = 27    'Keep editing until
                               '  user presses Esc
.
.
.
```

When the user finally presses **Esc**, the data entered into the form may be accessed by examining the contents of the Form$() string array.

EndOfForms

BASIC function contained in EDITFORM.BAS

- Purpose

  EndOfForms returns the number of the last field on any form.

- Syntax

```
LastFld% = EndOfForms%(Fld())
```

- Where

  LastFld%: The value of the last field on the form

  Fld(): Field information TYPE array (see *FLDINFO.BI*)

- Comments

  Because EndOfForms has been defined as a function, it must be declared before it may be used.

  This function can be used to determine the last field number on a form, and is particularly useful for a multi-page forms.

Evaluate

BASIC function contained in EVALUATE.BAS

- Purpose

  Evaluate is a full-featured expression evaluator that accepts a formula in an incoming string, and returns a double-precision result. Capitalization is ignored (in keywords such as LOG and SIN), except for the "E" used for scientific notation: to Evaluate, a lowercase "e" represents the constant, and an uppercase "E" is for the exponent.

- Syntax

```
Answer# = Evaluate#(Expression$)
```

- Where

  Expression$: A string containing a mathematical expression, with optional parentheses, operation keywords (such as ABS or SIN), and numbers; if the string expression is invalid, the string is returned in Expression$ with a leading percent sign (%) appended

  Answer#: Receives the computed answer

- Comments

  Because Evaluate has been defined as a function, it must be declared before it may be used.

  Scientific notation is supported using "E" (but not "e", "D" or "d"). What follows is a list of operations supported by Evaluate:

ABS	Absolute Value
AND	Logical AND
ARCCOS	Arc Cosine
ARCCOSH	Arc Hyperbolic Cosine
ARCCOT	Arc Cotangent
ARCCOTH	Arc Hyperbolic Cotangent
ARCCSC	Arc Cosecant
ARCCSCH	Arc Hyperbolic Cosecant
ARCTANH	Arc Hyperbolic Tangent
ARCSEC	Arc Secant
ARCSIN	Arc Sine
ARCSINH	Arc Hyperbolic Sine
ATN	Arc Tangent
CLG	Common Log (base 10, what LOG really is)
COS	Cosine

COT	Cotangent
CSC	Cosecant
CSCH	Hyperbolic Cosecant
EXP	Exp
LOG	Natural Log (base e, what BASIC calls LOG)
NOT	Logical NOT
OR	Logical OR
SINH	Hyperbolic Sine
SECH	Hyperbolic Secant
SEC	Secant
SIN	Sine
SQR	Square Root
TAN	Tangent
TANH	Hyperbolic Tangent

The following list shows math operators supported by Evaluate:

!	Factorial
^	Exponentiation
*	MultiplicationR
/	Division
\	Integer Division
+	Addition
-	Subtraction (or unary minus, such as -15)
<	Less than
=	Equal to
>	Greater than

- Example

```
X = EVALUATE("10 * (12^3+(4E-13))/LOG(8)")
```

Exist

assembler function contained in GFORMS.LIB

- Purpose

  Exist will quickly determine the presence of a file.

- Syntax

```
There% = Exist%(FileName$)
```

- Where

  FileName$: File name or file specification

  There%: Assigned to -1 if FileName$ exists; 0 if FileName$ does not exist

- Comments

  Because Exist has been designed as a function, it must be declared before it may be used.

  The main purpose of Exist is to prevent the error caused by trying to open a file for input when it does not exist. Rather than having to set up an ON ERROR trap just prior to each attempt to open a file, Exist will directly tell if the file is present.

  In the past, programmers have tried to avoid an error by opening a file for random access, which does not cause an error. Then the BASIC LOF function would be used to see if the file's length is zero, meaning it was not there. The problem with that approach, besides being a lot of extra work—is that an empty file could be created in the process. For this reason, we recommend using the Exist function.

  It is important to know that FileName$ may optionally contain a drive letter, a directory path, and either of the DOS wild card characters.

- Example

  This example returns -1 if there are .BAS files on the \STUFF directory of the B drive:

```
There% = Exist%("B:\STUFF\*.BAS")
```

FGet

assembler subroutine contained in GFORMS.LIB

- Purpose

  FGet reads data from a disk file in a manner similar to BASIC's binary GET command, but it returns an error code rather than requiring the use of ON ERROR.

- Syntax

```
CALL FGet(Handle%, Destination$)
```

- Where

  Handle%: Handle assigned when the file was opened

  Destination$: String that is to receive the data; the length of Destination$ determines how many bytes are to be read

- Comments

  FGet reads data from the specified file at the location held in the DOS file pointer. The current pointer location is established by the most recent read or write operation or by using the supplied FSeek routine.

  The length of Destination$ is used to tell FGet how many bytes it is to read to ensure that sufficient room has been set aside. If FGet had been written to expect a separate variable to specify the number of bytes, it would be possible to corrupt string memory by failing to first assign the string to a sufficient length.

  Only two errors are likely when using FGet: either the DOS handle number was invalid, or the destination string was null.

- Example

  This example gets one byte of information from the current file, at the location specified by the DOS file pointer.

```
X$ = SPACE$(1)          'set one byte aside
CALL FGet(Handle%, X$)  'read the byte value from
                        '  the file
```

FixDate

BASIC subroutine contained in EDITFORM.BAS

- Purpose

  FixDate changes the format of a date string.

- Syntax

  ```
  CALL FixDate(Dat$)
  ```

- Where

  Dat$: String containing the date in a variety of string formats

- Comments

  This subprogram ensures that dates are formatted in a consistent manner. For instance, FixDate forces all months and days to have two numerical digits (single-digit months or days will have a leading zero). It also ensures that a century is entered as two digits. Thus, "3-4-91" will become "03-04-1991" after calling FixDate.

- Example

  ```
  CALL FixDate(Dat$)
  ```

FldNum

BASIC function contained in EDITFORM.BAS

- Purpose

  FldNum returns the field number corresponding to a specified field name.

- Syntax

```
FldNumber% = FldNum%(FldName$, Fld())
```

- Where

  FldNumber%: Number of the field named by FldName$

  FldName$: String containing the field name

  Fld(): Field information TYPE array (see *FLDINFO.BI*)

- Comments

  Because FldNum has been defined as a function, it must be declared before it may be used.

  FldNum makes it easy to obtain the number of a field if all you have available is its name. The routine is useful for creating programs which do not have to be modified as your data entry form changes. It also makes source code more intelligible by allowing long variable names to refer to short field names.

- Example

  This example finds which field is named "DISCRATE" (the discount rate). Then, the field number is used to obtain the value of a field.

```
DiscountRateFld = FldNum("DISCRATE", Fld())
DiscountRate = VAL(Form$(DiscountRateFld, 0))
```

FOpen

assembler subroutine contained in GFORMS.LIB

- Purpose

  FOpen is used to open a disk file in preparation for reading or writing using the FGet or FSeek routines.

- Syntax

  ```
  CALL FOpen(FileName$, Handle%)
  ```

- Where

  FileName$: Name of file to be opened

  Handle%: Handle assigned by DOS for all subsequent access; if an error occur when trying to open the file, Handle% returns 0

- Comments

  FOpen will open any file and will also accept an optional drive or directory as part of the file name. However, it will not create a file. If you are not sure whether a file exists you should first use the Exist function.

  It is up to your program to store the handle number that DOS assigns and to use that handle whenever you access the file again.

- Example

  This example opens the file MYFORM.FRM and assigns an integer file handle number to it.

  ```
  CALL FOpen("MYFORM.FRM", Handle%)
  ```

GArraySize

BASIC function contained in GARRAYSZ.BAS

- Purpose

  GArraySize returns the number of bytes required by BASIC's graphic GET statement to hold a specified region of the screen.

- Syntax

```
Size& = GArraySize&(X1, Y1, X2, Y2)
```

- Where

  X1 and Y1 define the upper-left corner and X2 and Y2 define the lower right corner of the region to be saved.

- Comments

  GArraySize& can be used in any BASIC supported graphics mode.

- Example

```
REDIM IntArray%(GArraySize&(45, 25, 500, 125) \ 2)
```

GetFldDefG

BASIC subroutine contained in FRMFILE.BAS

- Purpose

  GetFldDefG retrieves information from a form file and places it in a structure for later reference by other routines. It also loads formulas and help messages into the Form$() data array.

- Syntax

```
CALL GetFldDef(FrmName$, StartEl%, Fld(), Form$())
```

- Where

  FrmName$: Name of the form (.FRM) definition file

  StartEl%: Starting element in the Fld() array in which the form information is to be loaded

  Fld(): Field information TYPE array (see *FLDINFO.BI*)

  Form$(): Form string array (see *Form$() array*)

- Comments

  This routine allows a calling program to load a .FRM file so that it may be properly processed by EditFormG. The NumFieldsG function should be used before this routine in order to properly dimension the Fld() and Form$() arrays.

- Example

  An example of this routine is shown in the section *DemoAnyG.BAS* under *Performing Data Entry*.

GetGMP

BASIC subroutine contained in GETGMP.BAS

- Purpose

  GetGMP loads .GMP image files from disk into an array.

- Syntax

```
GetGMP(Name$, GMPFile$, Array%(), ErrCode%)
```

- Where

  GMPFile$: The name of the GMP file to display; the GMP extension is not required

  Array(): An integer array that will be used to hold the image (rediminsioned to 0 before the call)

  ErrCode%: Returns a value that indicates whether or not the image was loaded successfully; a value of 1 indicates that an error occurred opening the file; a value of 2 indicates that the file was not found

- Comments

  GetGMP lets you display .GMP files created with the **Save Paste Buff...** option from the **File** menu from your own programs.

- Example

  Before calling the GetGMP subroutine, you must first create an Array() to hold the image:

```
REDIM Array(0)
CALL GetGMP("Pencil", Array(), ErrCode)
```

  After the image is loaded, it can be placed anywhere on the screen with BASIC's graphic PUT command:

```
PUT (10, 10), Array, PSET
```

GetRec

BASIC subroutine contained in RANDOMG.BAS

- Purpose

  GetRec retrieves a specified record and any associated notes from a database.

- Syntax

```
CALL GetRec(RecNo&, Form$(), Fld())
```

- Where

  RecNo&: Record number to retrieve

  Form$(): Form string array (see *Form$() array*)

  Fld(): Field information TYPE array (see *FLDINFO.BI*)

- Comments

  GetRec loads the specified record into the form buffer held in Form$(0, 0). Once this is done, it is necessary to call the UnPackBuffer routine so that the remaining elements in the Form$() array are properly filled.

  The data which is read by this routine is expected to be in a file with a .DAT extension, while any associated notes fields are read from a .NOT notes file.

  Usually, the OpenFiles routine is called before using either GetRec or SaveRec.

- Example

  See the section *Random Access File Operations* for more information about using the routine.

GMove2VE

assembler subroutine contained in GFORMS.LIB

- Purpose

  GMove2VE will save and restore any rectangular region of the screen to a video memory location which you specify.

- Syntax

```
DestSegment% = &HA800
CALL GMove2VE (BYVAL FromCol%, BYVAL FromLine% _
  BYVAL Cols%, BYVAL Lines%, BYVAL DestSegment%,_
  BYVAL Direction%)
```

- Where

  FromCol%: The upper left column (in text columns) of the region to be moved

  FromLine%: The upper left row (in pixels) of the region to be moved

  Cols%: The width of the region to be moved (in text columns)

  Lines%: The height of the region to be moved (in pixels)

  DestSegment%: Provides the routine with a location to send the information; the segment value should be within the range of EGA or VGA graphics memory available

  Direction%: Specifies whether the image will be saved or restored; a zero saves the image, any other value will restore the image.

- Comments

  This routine uses screen memory to store the image. This approach has two advantages: Graphics saves and restores require one-fourth the instructions of other save and restore routines, and graphics memory is often not used and is therefore less costly to the programmer than using general memory.

  All parameters for this routine are passed by value to provide the maximum speed.

- Example

  The following example saves and restores the upper-left corner 10 column by 100 lines region of the screen.

```
DEFINT A-Z
DECLARE SUB GMove2VE (BYVAL FromCol%, BYVAL _
```

```
    FromLine%, BYVAL Cols%, BYVAL Lines%, BYVAL _
    DestSegment%, BYVAL Direction%)
SCREEN 12                   'sets the monitor in VGA mode

LINE (0,0) - (79,99), 1, B


'save the image
CALL GMove2VE (1, 0, 10, 100, &HAA00,0)

CLS
'restore the image
CALL GMove2VE (1, 0, 10, 100, &HAA00, -1)
```

The beginning of the EGA's high-resolution second screen starts at &HA800. On the EGA display there is a 128K free for the storage of images.

The VGA high-resolution mode doesn't have a second screen per se, but you still can use this routine. For the VGA, the first unused graphics segment would be &HAA00 as shown in the above code example. The VGA has only 96K available memory for the storage of images.

The following formula will help to calculate the amount of memory used by an image saved with this routine:

```
MemUsed% = Cols% * Lines% * 4
```

To determine the next segment where graphics images can be stored, use

```
NextSegment% = ThisSegment& + MemUsed% \ 16 + 1
```

where ThisSegment% is the segment where the current graphics image is being stored.

This routine is a vital part of saving graphics images for use in the graphics GQSMenu menu and in the ListBox routine.

GMove4VE

assembler subroutine contained in GFORMS.LIB

- Purpose

  GMove4VE will save and restore any rectangular region of the screen to an array you specify.

- Syntax

```
CALL GMove4VE (BYVAL FromCol%, BYVAL From Line%,_
  BYVAL Cols%, BYVAL Lines%, BYVAL DestSegment%, _
  BYVAL Direction%)
```

- Where

  FromCol%: The upper left column (in text columns) of the region to be moved

  FromLine%: The upper left row (in pixels) of the region to be moved

  Cols%: The width of the region to be moved (in text columns)

  Lines%: The height of the region to be moved (in pixels)

  DestSegment%: Provides the routine with a location to send the information. This segment value is determined by finding the segment of a pre-dimensioned array. The segment of an array can be found as follows:

```
REDIM Array%(0 to 5000)
DestSegment% = VARSEG(Array(0))
```

  Direction%: Determines whether the image will be saved or restored; a value of zero saves the image, any other value will restore the image.

- Comments

  All parameters for this routine are passed by value to provide the maximum speed.

  The memory location must be declared prior to saving the image into the array. To calculate the amount of memory required use the following formula:

```
MemoryNeeded% = ColumnsUsed% * LinesUsed% * 4 + 4
```

  Once the amount of memory required has been calculated, you will dimension an integer array with half of the elements contained in Memory-

must then run BASIC with the /Ah parameter and compile your programs with this parameter as well. In addition, you will need to create and pass this routine a long integer array where each element will provide 4 bytes of memory space. To make an array holding 128K of memory, dimension it as follows:

```
REDIM LongArray&(0 to 32767)
```

- Example

The following example saves and restores a region 10 columns wide by 100 lines high in the upper-left hand corner of the screen.

```
DEFINT A-Z
DECLARE SUB GMove4VE(BYVAL Col, BYVAL ScrnLine,
   BYVAL Cols, BYVAL DestSegment, BYVAL Direction)
SCREEN 12                'sets the monitor in VGA mode

LINE (0,0)-(79,99), 1 B
'save the image
MemNeeded% = 10* 100 * 4 + 4
DIM A%(MemNeeded% \2)   'each integer counts for 2
                        ' bytes
CALL GMove4VE(1,0, 10, 100, VARSEG(A%(0)), ))
WHILE INKEY$ = "":WEND

CLS
'restore the image
CALL GMove4VE (1, 0, 10, 100 VARSEG(A%(0)), -1)
```

GPrint0VE

assembler subroutine contained in GFORMS.LIB

- Purpose

  GPrint0VE prints a string on the 16-color EGA and VGA high-resolution graphics screens in a specified color.

- Syntax

```
CALL GPrint0VE (BYVAL Row%, BYVAL Column%, Text$, _
  BYVAL TextColor%)
```

- Where

  Row% and Column%: The normal coordinates used by the BASIC LOCATE statement

  Text$: Any text string

  TextColor%: Holds the combined foreground and background colors; the following formula can be used to set the colors used:

```
TextColor% = Foreground% + (Background * 256)
```

- Comments

  Numeric parameters for this routine are passed by value to provide the maximum speed.

- Example

  The following example shows how to print a string to the VGA using the color blue for the foreground, and the color gray for the background.

```
DEFINT A-Z
DECLARE SUB GPrint0VE (BYVAL Row, BYVAL Col, BYVAL _
  Column, Text$, BYVAL TextColor)
SCREEN 12                       'sets the monitor in
                                '  VGA mode

GPrint0VE 1, 10, "This is on row 1", 1 + 7 * 256
```

  This routine is many times faster than the BASIC PRINT statement. It also allows you to specify a background color for the text string.

HideCursor

assembler subroutine contained in GFORMS.LIB

- Purpose

  HideCursor turns off the mouse cursor.

- Syntax

  ```
  CALL HideCursor
  ```

- Comments

  Any program that is to be "mouse aware" will need to turn on the mouse cursor before expecting a user to access the mouse. Likewise, it is only common courtesy to turn it off again before returning them to the DOS prompt. Also, for graphics programming, you must turn the mouse off before drawing something on the screen.

  One very important point to be aware of regarding the HideCursor routine is how the current on and off status is maintained internally by the mouse driver. Unlike the normal text cursor that is turned on or off with the BASIC LOCATE command, the mouse cursor keeps track of how many times it was turned on or off. Thus, if you call HideCursor twice in a row, you will need to call ShowCursor twice before it will be visible again.

  In graphics mode, when you want to draw something at the location of the mouse, it is necessary to turn off the mouse cursor temporarily while you are drawing. In graphics mode, the mouse has a copy of the screen image beneath itself. If you draw over the cursor with the cursor on, when the cursor moves, the mouse driver will re-draw the previous image, without what you drew.

  This is why you see the mouse flicker in large graphics applications. These applications turn the mouse off and on many times while drawing to the screen. It is for this reason that the above mentioned characteristic of the HideCursor routine can be useful. If you have multiple routines drawing graphics on the screen, it is necessary that each routine turn the mouse cursor off before drawing and turn it back on before leaving. However, due to the nature of graphics programming, a routine cannot always expect to be called from another routine which has previously turned off the mouse. For example, a routine designed to draw an entire face might call a routine to draw an eye. If the eye routine were to be called separately, it should turn off the mouse cursor itself. If it is called from within another routine which has already turned off the cursor, then it should not turn on the cursor when it is finished. Instead the count maintained by the mouse

driver is merely decremented when the eye routine calls ShowCursor to turn the cursor back on.

- See Also

  ShowCursor.

InitMouse

assembler subroutine contained in GFORMS.LIB

- Purpose

  InitMouse is used to determine if a mouse if present in the host PC, and to reset the mouse driver software to its default values.

- Syntax

  ```
  CALL InitMouse(HaveMouse%)
  ```

- Where

  HaveMouse%: Receives -1 if a mouse is present, or 0 if no mouse is installed

- Comments

  Because InitMouse resets the mouse driver values (the mouse cursor color, its travel range and sensitivity, etc.), it would probably be called only once at the start of a program.

  Understand that InitMouse doesn't actually detect the physical presence of the mouse hardware. Rather, the mouse driver software must be installed before a mouse will be detected. Newer versions of Microsoft's mouse driver software actually detect if the mouse is physically attached to the machine, and will not load the driver unless the mouse is connected.

KeyDown

assembler function contained in GFORMS.LIB

- Purpose

  KeyDown reports if any keys are currently being pressed.

- Syntax

```
KeyIsDown = KeyDown%
```

- Where

  KeyIsDown: Returns -1 (True) if a key is currently being pressed, or 0 if no keys are pressed

- Comments

  Because KeyDown has been designed as a function, it must be declared before it may be used. KeyDown must also be installed before it will operate, and this is done by calling the InstallKeyDown routine.

  KeyDown is useful in a variety of situations. It is used by the EditFormG subroutine to detect when a key that was pressed to activate a push button or mouse field has been released.

  In order to detect when keys are pressed and released, KeyDown takes over the keyboard interrupt. This is why it must be installed. KeyDown automatically removes itself from the interrupt chain automatically when your program ends.

  However, a bug in QBX (the QB editor that comes with BASIC 7 PDS) prevents the automatic de-installation from working correctly. Therefore, you must call DeinstallKeyDown manually before ending your program if you are using QBX. De-installing is not necessary with QuickBASIC 4.0 or 4.5, nor with programs that are compiled to .EXE files.

  Note that when multiple keys are pressed (such as Alt-F), Keydown returns -1 when Alt is first pressed. But as soon as either combination key is released KeyDown returns 0.

- Example

  See EditFormG for an example of using KeyDown.

LibGetFldDefG

BASIC subroutine contained in LIBFILE.BAS

- Purpose

  LibGetFldDefG retrieves information from a form file contained in a custom .GSL library and places it in a structure for later reference by other routines. It also loads formulas and help messages into the Form$() data array.

- Syntax

```
CALL LibGetFldDef(LibName$, FrmName$, StartEl%, _
  Fld(), Form$(), ErrCode%)
```

- Where

  LibName$: Name of the custom .GSL library; the .GSL extension is assumed and does not need to be entered

  FrmName$: Name of the form (.FRM) definition file

  StartEl%: Starting element in the Fld() array in which the form information is to be loaded

  Fld(): Field information TYPE array (see *FLDINFO.BI*)

  Form$(): Form string array (see *Form$() array*)

  ErrCode%: Returns a value that indicates whether or not the form definition was loaded successfully; a value >1 indicates that an error occurred opening the library file; a value of 1 indicates that the file was not found in the specified library

- Comments

  LibGetFldDefsG is used to replace the GetFldDefsG subroutine when the form definition file is stored in a custom .GSL library. Note that the calling syntax for LibGetFldDefsG is identical to that used by GetFldDefsG except for the addition of the LibNames$ and ErrCode% arguments and the need to include the .FRM extension in FrmName$.

- Example

  This example loads the form information for "MyForm" from a library named "MyLib.GSL".

```
CALL LibGetFldDefsG("MyLib", "MyForm.FRM", 0,
Fld(), Form$(),ErrCode%)
```

LibGetGMP

BASIC subroutine contained in LIBFILE.BAS

- Purpose

  LibGetGMP loads .GMP files from a custom .GSL library into an array.

- Syntax

```
LibGetGMP(LibName$, GMPFile$, Array%(), ErrCode%)
```

- Where

  LibName$: Name of the GSL library; the .GSL extension is assumed and does not need to be entered

  GMPFile$: The name of the GMP file to display; the .GMP extension must be included

  Array(): An integer array that will be used to hold the image (rediminsioned to 0 before the call)

  ErrCode%: Returns a value that indicates whether or not the image was loaded successfully; a value > 1 indicates that an error occurred opening the library file; a value of 1 indicates that the file was not found in the specified library

- Comments

  LibGetGMP is used to replace the GetGMP subroutine when the .GMP files are read from a custom .GSL library. Note that the calling syntax used by LibGetGMP is identical to that for the GetGMP subroutine except for the addition of the LibName$ argument and the need to include the .GMP extension in GMPFile$.

- Example

  Remember that before calling the LibGetGMP subroutine, you must first create an Array() to hold the image:

```
REDIM Array(0)
CALL LibGetGMP(MyLib$, "Pencil.GMP", Array(), _
   ErrCode)
```

  After the image is loaded, it can be placed anywhere on the screen with BASIC's graphic PUT command:

```
PUT (10, 10), Array, PSET
```

LibNumFieldsG

BASIC function contained in LIBFILE.BAS

- Purpose

  LibNumFieldsG returns the number of fields in a form contained in a custom .GSL library.

- Syntax

```
N% = LibNumFieldsG%(LibName$, FormName$)
```

- Where

  LibName$: Name of the GSL library; the .GSL extension is assumed and does not need to be entered

  N%: Number of fields in FormName$; if an error occurs, N% is set to -1

  FormName$: String containing the full path and file name of the form definition file; the .FRM extension must be included

- Comments

  LibNumFieldsG is used to replace the NumFieldsG function when the form definition file is stored in a custom .GSL library. Note that the calling syntax for LibNumFieldsG is identical to that used by NumFieldsG except for the addition of the LibNames$ argument and the need to include the .FRM extension in FormName$.

  LibNumFieldsG is used to dimension the Fld() TYPE array and Form$() data array to the proper number of elements before calling LibGetFldDefG.

  Because LibNumFieldsG has been defined as a function, it must be declared before it may be used.

- Example

  This example returns the number of fields in the MYFORM.FRM file:

```
NumFields% = LibNumFieldsG%("MyLib", "MYFORM.FRM")
```

LibShowForm

BASIC subprogram contained in LIBFILE.BAS

- Purpose:

  LibShowForm displays any .PCX screen that you design with the Graphics QuickScreen screen editor and is stored in a custom .GSL library. It sets the proper screen mode, number of text rows, and adjusts the color palette. The row and column arguments allow you to position partial screen images.

- Syntax:

```
Call LibShowForm (LibName$, FileName$, Fld(), _
  Row%, Col%, VPage%, ErrCode%)
```

- Where:

  LibName$: The name of the custom .GSL library; the .GSL extension is assumed and does not need to be entered

  FileName$: The name of the .PCX file to be displayed; the .PCX extension must be included

  Fld(): An array containing the form's field definitions; the array is loaded using either the LibGetFldDefG subroutine or by using the *FileName** BASIC subroutine that is created when you save your forms; the Fld() array is used to pass the following information:

  Fld(0).Value: Contains the screen mode to use when displaying the screen; the value is either 5 for EGA SCREEN 9, or 8 indicating VGA SCREEN 12

  Fld(0).Indexed: Contains the text height in pixels of the required ROM font (8, 14 ,16); this value along with the screen mode tells LibShowForm how many text rows to set.

  If you are repositioning a partial .PCX image that has field definitions, the Fld(N).Row, Fld(N).LCol, Fld(N).RCol, and Fld(N).ScratchI variables are automatically updated for each field to their new coordinates.

  Row: The Y screen coordinate (in pixels) for the upper left corner of a partial screen; any valid screen row may be specified as long as the bottom-most field is fully displayed; to display a full screen image set this value to 0

Col: The upper left column for displaying a partial screen image; any valid screen column may be specified as long as the right-most field is fully displayed; to display a full screen image set this value to 0

VPage: Specifies which video page to use when loading an EGA (640x350) image; VPage set to 0 loads the image to the visible video page; VPage set to 1 loads the image to the background video page where it can be restored to the visible page with one of the wipe routines contained in EGAWIPES.BAS; the VPage% parameter is ignored by VGA (640x480) screens

ErrCode: Returns a value that indicates whether or not the image was loaded sucessfully; a value of 1 indicates that an error occurred when opening the library file; a value of 2 indicates that the specified file was not found in the library; a value of 3 means that you attempted to display the file an an incompatible monitor, and a value of 4 indicates that an error occurred while loading the screen.

* A BASIC module and subroutine are optionally created when you save your screens using the same name as your form.

- Comments:

LibShowForm is used to replace the ShowForm subroutine when the .PCX files are read from a custom .GSL library. Note that the calling syntax for LibShowForm is identical to that used by ShowForm except for the addition of the LibNames$ argument and the need to include the .PCX extension in FileName$.

LibShowForm can be used to display any EGA 640x350 or VGA 640x480 16-color .PCX file that is stored in a custom .GSL library. If you are displaying a .PCX file with no field definitions, the Fld() array can be dimensioned to 0 elements. If all values in the Fld() array are set to 0, the screen mode will be set and the image displayed in whatever screen mode is indicated by the value of GPDat%(31). You can also force a specific screen mode for display-only screens by assigning appropriate values to Fld(0).Value and Fld(0).Indexed.

GPDat%(31) is set in the SETUP.BAS $INCLUDE file to indicate the current monitor type. See Appendix A for more information on the GPDat%() array.

■ Example

This example displays the "Screen1.PCX" image stored in "Custom.GSL".

```
CALL LIBShowForm("Custom", "Screen1.PCX", _
  Fld(), 0, 0, 0, ErrCode%)
```

ListBox

BASIC subroutine contained in LISTBOX.BAS

- Purpose

  ListBox is a comprehensive menu subprogram with many important capabilities including full support for a mouse. It can optionally save the underlying screen to conventional or video memory. If there are more choices than can be displayed in the specified number of rows, a scroll bar will be added to the menu. ListBox is used to support multiple-choice fields in a form and is also used in the Graphics QuickScreen **File Open**... dialog box.

- Syntax

```
CALL ListBox(Item$(), Choice%, MaxLen%, Rows%, Ky$, _
  Hk%(), Action%)
```

- Where

  Item$: Conventional (not fixed-length) string array containing the list of menu choices.

  Choice%: Indicates which choice was selected, and may also be pre-loaded to force a given choice to be highlighted initially

  MaxLen%: Maximum length of any menu choice, thus establishing the menu width (choices longer than MaxLen% will be displayed truncated)

  Rows%: The number of lines to display in the ListBox

  Ky$: Holds the last key that was pressed by the user

  Hk%(): Integer array of hotkeys; this array is used only by the dialog box routine in Graphics QuickScreen and should be dimensioned to 0

  Action%: Tells how ListBox should be used (see comments below)

- Comments

  Displaying a list of items in a window is only one of the features of ListBox. Its real power comes from the use of the Action variable, and the its ability to be polled.

  The Action variable has nine different possible settings that tell ListBox how it is to behave. Each of the possible Action values is described below.

If Action is set to zero, then the menus will operate the way you would expect a "normal" menu to work. That is, the menu is displayed, and an INKEY$ loop repeatedly waits for the user to press a key or a mouse button. Once a key or mouse button has been pressed, control is returned to the calling program. The Choice variable may then be examined to see what selection the user chose.

When Action is set to -3, ListBox simply displays the menu without highlighting a choice and returns control immediately to the calling program.

When Action is set to -2, -1, or 1, ListBox displays the menu and highlights the specified choice. Action set to -1 first saves the underlying screen to conventional memory while Action -2 saves the screen to video memory. Action set to 1 does not save the underlying screen. Control is returned to the calling program immediately, however Action is set to 3 for subsequent calls. Since Action set to 3 is how you will be polling the menu subsequently, this saves you an extra step.

Setting Action to 2 lets you redisplay the menu, in those cases where you may wish to change the contents of the menu without having to first exit ListBox and call it again with the new selections. Action 2 also resets itself to 3 for subsequent calls. If the menus are called with Action equal to 3, the keyboard and mouse are merely polled to see if a key or button has been pressed.

If Action is still set to 3 when the menu returns, it means that no keys or mouse buttons were pressed.

If Action is returned set to 4, the user either made a selection or pressed Escape. In this case, the Choice and Ky$ variables should be examined.

If Action is set to 5, ListBox will remove itself and restore the original screen if it had been called initially with Action = -1 or Action = -2.

Use Locate to position the upper-left corner of the listbox.

If you are not using multiple-choice fields you should use the "blank" ListBox subprogram which is part of NOMULTG.BAS. This resolves references to ListBox without needing to load the full ListBox source code.

ListBox also requires the SCROLLB.BAS module to manage the scroll bar.

Message

BASIC subprogram contained in FORMEDIT.BAS

- Purpose

  Message displays text messages in a box

- Syntax

```
CALL Message(Msg$, Row)
```

- Where

  Msg$: Message string

  Row: Top screen row of the message box

- Comments

  This routine quickly displays any text you supply in a conventional (variable-length) string. When you call Message with a non-null string, the message is displayed in a box or "window". If you call Message again with a null string, the window previously displayed is removed, and the screen image underneath the message will be restored.

- Example

```
CALL Message("This is a help message.", 10)
```

Colors for the various components of the message box are set in GPDat%() elements 96 - 99:

```
GPDat%(96) = Background color  'Default = 7 gray
GPDat%(97) = Text color        'Default = 0 black
GPDat%(98) = Highlight color   'Default = 15 white
GPDat%(99) = Shade color       'Default = 8 Dk.gray
```

Motion

assembler subroutine contained in GFORMS.LIB

- Purpose

  Motion allows a program to establish the sensitivity of the mouse cursor motion.

- Syntax

```
CALL Motion(Value%)
```

- Where

  Value%: The desired sensitivity ranging between 1 and 32767, with 1 being the most sensitive

- Comments

  Even though the mouse driver software allows setting the horizontal and vertical sensitivity separately, Motion uses the same value for both. This seems to be the most logical way to control a mouse, while eliminating yet another passed parameter.

  The stated upper range for the motion sensitivity is 32767, however values beyond 100 or so are hopelessly insensitive.

  You may be interested to know that Microsoft calls the unit of cursor distance for the mouse a "Mickey".

MultMonitor

assembler function contained in GFORMS.LIB

- Purpose

  MultMonitor% makes it east to determine the type of display adapter currently active.

- Syntax

```
M% = MultMonitor%
```

- Where

  MonType%: Integer value representing the detected monitor type currently in use; a value of 0 means no graphics monitor is attached.

Bit	Value	Meaning
0	1	Hercules adaptor is attached
1	2	CGA capable adaptor attached
2	4	mono EGA adaptor is attached
3	8	color EGA adaptor is attached
4	16	mono VGA adaptor is attached
5	32	color VGA adaptor is attached
6	64	mono MCGA adaptor is attached
7	128	color MCGA adaptor is attached
8	256	EGA adaptor emulating CGA
9	512	IBM 8514/A adaptor is attached

  For example, a system which has both a VGA color monitor and a Hercules monitor connected will return a value of 33 (32 for VGA + 1 for Hercules).

- Comments

  To check if a VGA monitor exists, use the following line of code:

```
IF (M% AND 32) <> 0 THEN PRINT "Can use VGA"
```

- Example

  See SETUP.BAS for an example of using MultMonitor.

NumFieldsG

BASIC function contained in FRMFILE.BAS

- Purpose

  NumFieldsG returns the number of fields in a form.

- Syntax

```
N% = NumFieldsG%(FormName$)
```

- Where

  N%: Number of fields in FormName$

  FormName$: String containing the full path and file name of the form

- Comments

  Because NumFieldsG has been defined as a function, it must be declared before it may be used.

  This function is used to dimension the Fld() TYPE array and Form$ data array to the proper number of elements before calling GetFldDefG.

- Example

  This example returns the number of fields in the MYFORM.FRM file:

```
NumFields% = NumFieldsG%("MYFORM.FRM")
```

Num2Date

assembler function contained in GFORMS.LIB

- Purpose

  Num2Date converts a previously-encoded integer date to an equivalent date string.

- Syntax

```
D$ = Num2Date$(Days%)
```

- Where

  D$: Formatted date string (MM-DD-YYY)

  Days%: Integer value from -29219 to 31368

- Comments

  Because Num2Date has been designed as a function, it must be declared before it may be used.

  Please see the Date2Num discussion and example for more information.

OpenFiles

BASIC subprogram contained in RANDOMG.BAS

- Purpose

  OpenFiles is used to open a random access database (.DAT) file, and field-format it to the data buffer Form$(0, 0). If there are multi-line notes fields contained in the form, a Notes database file (.NOT) is also opened.

- Syntax

```
CALL OpenFiles(FormName$, Form$(), Fld() AS _
  FieldInfoG)
```

- Where

  FormName$: Base name of the database file to open (without the .DAT extension)

  Form$(): Form string array (see *Form$() array*)

  Fld(): Field information TYPE array (see *FLDINFO.BI*).

- Comments

  OpenFiles looks in the current directory for the form name you provide. If you want to access files on a different drive or directory, then you must append the path in front of the FormName$.

  If the form is found, it and its associated notes file are opened; if the form file is not found then it is created.

  Once the random file is open, Form$(0, 0) is fielded to match the fields as needed. Fld(0).RelHandle holds the handle for the .DAT file that was opened, and Fld(0).ScratchI hold the handle for the Notes file if one was opened.

  The GetRec and SaveRec routines can be used after OpenFiles has been successful.

OpenPCXFile

assembler subroutine contained in GFORMS.LIB

- Purpose

  OpenPCXFile% opens the specified PCX file, and loads the header information, including palette information, into the string specified.

- Syntax

```
Array$ = SPACE$(68 + 768)
Success% = OpenPCXFile%(Filename$, Array$)
```

- Where

  Filename$: A string containing the name of the PCX file

  Array$: A string of length (68 + 768); the first 68 bytes receive the header information. If the file is a 256-color PCX file, then the information contained in the last 768 bytes of this string are the palette information for the 256-color mode.

- Comment

  After opening the PCX file, the image can be displayed by calling DISPPCXVE:

```
Array$ = Space$ (68+768)
IF NOT OpenPCXFile% (FileName$, Array$) THEN
  EXIT SUB
END IF
CALL DISPPCXVE(VideoPage%)
```

PositionPCXVE

assembler subroutine contained in GFORMS.LIB

- Purpose

  PositionPCXVE is used to locate a .PCX image which is loaded with the DispPCXVE routine. By calling this routine immediately prior to the DispPCXVE routine, a PCX image can be located at any column and line combination as defined by the Mixed coordinate system.

- Syntax

```
CALL PositionPCXVE (BYVAL LineStart%, BYVAL _
  ColStart%)
```

- Where

  LineStart%: Upper-left row where the image is to be displayed (in pixels)

  ColStart%: Upper-left column where the image is to be displayed (in text columns)

- Comments

  This routine is used in ShowForm to display partial PCX images.

- Example

  See ShowForm for an example of using PositionPCXVE

PrintArray

BASIC subprogram contained in EDITFORM.BAS

- Purpose

  PrintArray refreshes the screen by redisplaying the contents of fields in the form.

- Syntax

```
CALL PrintArray(FirstFld%, LastFld%, Form$(), Fld())
```

- Where

  FirstFld%: Starting field to be redisplayed

  LastFld%: Ending field to be redisplayed

  Form$(): Form string array (see *Form$() array*)

  Fld(): Field information TYPE array (see *FLDINFO.BI*)

- Comments

  Sometimes it is necessary to alter the contents of a field manually in a program. Refreshing the screen ensures that the user is aware of the new contents of each field on the form.

- Example

  This example refreshes only the third and fourth field of the currently-displayed form:

```
CALL PrintArray(3, 4, Form$(), Fld())
```

QEdit

BASIC subprogram contained in GQEDITS.BAS

- Purpose

  QEdit is a graphics mode text editor subprogram that may be called as a pop-up from within a BASIC program.

- Syntax

```
CALL GQEdit(Array$(), Ky$, Action%, Ed)
```

- Where

Array$():	Conventional (not fixed-length) string array that will hold the text being entered or edited
Ky$:	Returns holding the last key pressed
Action%:	Indicates how QEdit is being invoked (see comments below)
Ed:	TYPE variable that controls QEdit (see comments below)

- Comments

  The QEdit editing window may be positioned anywhere on the screen, and sized to nearly any number of rows and columns. QEdit can optionally save the underlying screen and it may be used in the 25-, 30-, 43-, or 60-line screen modes. QEdit also supports word-wrap, a mouse, and horizontal/vertical scrolling.

  All of the standard editing keys are supported. For example, Home and End move to the beginning and end of a line; the PgUp and PgDn scroll the screen by pages; and Ctrl-PgUp and Ctrl-PgDn move to the first and last lines, respectively. The cursor may also be moved to the top or bottom of the edit window with the Ctrl-Home and Ctrl-End keys.

  Similar to the BASIC editor, QEdit uses the Ctrl-Left and Ctrl-Right arrow keys to move the cursor by words and Ctrl-Y to delete a line of text.

  The call for QEdit is fairly simple to set up. Your program will need to dimension a conventional (not fixed-length) string array to hold the lines of text. The size to which the string array is dimensioned dictates the maximum number of lines that may be entered.

  If you intend to present a blank screen to your user, then no additional steps are needed to prepare the array. If you already have text that is to be edited, it may be placed in the array before QEdit is called.

The text may also be sent to QEdit as a single long line in the lowest array element. In that case, it will be wrapped automatically before being presented for editing. If you intend to read files prepared by a word processor that places each paragraph on its own line (such as XyWrite), you will probably want to read each line into every other element in the string array. This will preserve the spacing between paragraphs, and can be accomplished as shown below:

```
.
.
.
OPEN X$ FOR INPUT AS #1       'open the file
CurLine = 1                   'set current line counter
WHILE NOT EOF(1)              'read until the end
  LINE INPUT Array$(CurLine) 'get a line
  CurLine = CurLine + 2       'skip over next line
WEND
CLOSE #1                      'close the file
.
.
.
```


Like ListBox, the current cursor location indicates where to position the upper-left corner of the editing window. Arguments passed to QEdit are then used to indicate the width and height of the window, the margins, colors, and so forth. Let's take a close look at each of these in turn. Here's the QEdit calling syntax, once again:

```
CALL QEdit (Text$(), Ky$, Action%, Ed)
```

The Text$() array holds the text to be edited, as described above.

Ky$ returns the key holding the last key pressed. For example, it will hold CHR$(27) if the user pressed Esc to exit QEdit.

The Action argument sets the operating mode for QEdit as follows:

Action = 0

Use the editor in a non-polled mode. QEdit will take control, and return only when the user presses the **Esc** key. If you do not intend to add features to QEdit or take advantage of its multitasking capability, you may set Action to 0 and simply ignore the remaining Action parameters described below.

Action = -2, -1, or 1

Initialize the editor for polled mode. The edit window will be drawn, and the text (if any) displayed. Control is returned to the caller immediately without QEdit checking the keyboard. The Action flag is also set to 3 automatically. Action values of -2 and -1 first save the underlying screen to either video or conventional memory* respectively.

Action = 2

Redisplay the edit window and text, but without resaving the underlying screen. Control is then returned to the caller immediately without checking the keyboard. As above, the Action parameter will be set to 3 automatically.

Calling QEdit with an Action value of 2 is useful when changing the window size or location, to force QEdit to redisplay the text at the new location.

Note that if word wrap is on, Actions -2, -1, 0, 1, and 2 will cause the text to be re-wrapped to the right margin specified in Ed.Wrap (see below).

Action = 3

This is the "idle state" for QEdit. Each time QEdit is called with this value, it checks the keyboard and acts on any keys that were pressed. It then returns to the caller.

While QEdit is being polled the caller may examine the Ky$ parameter to determine which, if any, keys were pressed. The members of the Ed TYPE structure can also be examined and changed. Note that if the calling program chan-

ges any of the Ed values, QEdit should be called again with an Action of 2 to redisplay the edit window.

Action = 5 — Restores the screen that was saved when QEdit was called with Action set to 1.

The Ed parameter is a TYPE structure defined as EditInfo in the file QEDITYPE.BI. All of the additional parameters for QEdit are contained in this structure. Therefore, you must include QEDITYPE.BI in your calling program, and assign the elements needed to establish the window size, colors, and so forth. Note that passing a pointer to a TYPE variable this way is much faster and more concise than passing all of these parameters as part of the call. The following is a list of the elements in the EditInfo structure.

Ed.Rows — This sets the number of rows to be displayed in the window. The default maximum number of lines for an EGA monitor is 25, or 30 for a VGA monitor. If WIDTH is used to set more screen lines before QEdit is called, then the window may occupy up to 43 (for EGA) or 60 (for VGA) lines.

Ed.Wide — This sets the number of columns (up to 80) that are displayed in the window.

Ed.Wrap — This sets the right margin for word wrapping. This is independent of the right-most visible column, and may be set to nearly any value (up to 255). If the right margin extends beyond the right edge of the window, QEdit will scroll the text to accommodate it. Word wrap may also be disabled entirely by setting Ed.Wrap to 0.

Ed.HTab — This sets the number of columns to move when **Tab** or **Shift-Tab** is pressed. This parameter will default to 8 if a value of zero is given.

Ed.AColor — This sets the color of the edit window. The value combines both foreground and background colors into a single integer and is defined as follows:

```
Ed.AColor = fgColor + bgColor * 256
```

Ed.Frame — This is not supported, and is included for compatibility with the text mode version of QEdit provided with other Crescent products.

The remaining parameters are intended to be read by your program, and do not have to be set before QEdit is called.

Ed.LSCol — This holds the current left screen column of the editable window.

Ed.LC — This holds the left-most column of text being displayed, which will be greater than 1 if text is scrolled to the right.

Ed.CurCol — This holds the current text column number of the cursor within the edit window, which is not necessarily the current screen column.

Ed.TSRow — This holds the top screen row of the editable window.

Ed.TL — Holds the topmost row of the displayed text, which will be greater than 1 if text has been scrolled down.

Ed.CurLine — This holds the current text line number of the cursor within the edit window, which is not necessarily the current screen row.

Ed.UlCRow
Ed.UlCCol
Ed.BrCRow
Ed.BrCCol
Ed.CBlock — These are not supported in this version

Ed.Presses — This indicates whether a mouse button has been pressed, but not handled by the editor. This information is for your program to use if you intend to handle mouse presses that occurred outside of QEdit. Since Ed.Presses is non-zero only in that situation, you would then examine the Ed.MRow and Ed.MCol parameters (see below) to know where the mouse cursor was when the button was last pressed.

Ed.MRow	This holds the row where the mouse cursor was at the time the button was last pressed or if it is currently being pressed.
Ed.MCol	This holds the column where the mouse cursor was at the time the button was last pressed or if it is currently being pressed.
Ed.Insert	This is used to determine the current insert state mode. This will be 1 if QEdit is currently in the overtype mode or -1 if inserting is active.
Ed.Changed	This is used to see if the text has been changed. This parameter will be set to -1 if any changes or additions have been made to the text; otherwise it will be 0. This lets you know whether the file needs to be saved or not. However, you must clear this variable once the text has been saved.
Ed.LCount	This holds the number of active lines in the text string array, so you can know how many array elements need to be written to disk when saving text.
Ed.MErr	This is an error flag that signals errors that occurred within the editor. Ed.MErr will be 1 if there is insufficient memory. This could be caused by running out of string space with a large document, or not having enough far memory.

* To store an entire 640x480 16-color VGA screen requires approximately 154k of memory. Most video boards supply 256k of video memory leaving 102k of unused memory. When QEdit is called with action = -1, the underlying screen is saved to this portion of memory. The underlying screen saved by QEdit must therefore require no more than 102k of memory for storage, limiting the maximum size of the edit window to about 2/3 of a full screen.

If you must save larger portions of the screen, you can call QEdit with action set to -2 which saves the underlying screen in conventional memory. Since the array that holds the underlying screen will in this case be greater than 64k, you must start BASIC and compile your program with /ah.

If you are working in SCREEN 9, most display adapters will have enough memory to hold two entire screens. You can therefore have QEdit occupy the entire screen and still save the underlying screen to video memory with Action set to -2. The advantage of saving to video memory instead of conventional memory is that the use of video memory has no affect on the amount of conventional memory available to your program.

SaveField

BASIC subprogram contained in EDITFORM.BAS

- Purpose

  SaveField validates and formats a field before placing it in the Form$(0, 0) form buffer. This routine is often used before calling the PrintArray routine.

- Syntax

```
CALL SaveField(FldNo%, Form$(), Fld(), BadFld%)
```

- Where

  FldNo%: Field to be examined

  Form$(): Form string array (see *Form$() array*)

  Fld(): Field information TYPE array (see *FLDINFO.BI*)

  BadFld%: Returns 0 when valid; -1 when invalid

- Comments

  This routine first validates the data in the field by checking high and low acceptable ranges for the data. If the data in the field is not valid (i.e., it is out of the allowable range for the field, or it is an invalid date), then the BadFld flag will be set to -1. If BadFld is returned as 0, then the data is valid and SaveField will have updated the contents of the Form$(0, 0) form buffer with the current field's data.

- Example

  This example validates data in field three before updating the field buffer:

```
CALL SaveField(3, Form$(), Fld(), BadFld%)
```

SaveRec

BASIC subprogram contained in RANDOMG.BAS

- Purpose

  SaveRec saves information from a form to a specified record in a .DAT data file. Multi-line notes fields, if present, are written to a notes file having a .NOT extension.

- Syntax

```
CALL SaveRec(RecNo&, Form$(), Fld())
```

- Where

  RecNo&: Record number to save

  Form$(): Form string array (see *Form$() array*)

  Fld(): Field information TYPE array (see *FLDINFO.BI*)

- Comments

  The data currently in the form buffer Form$(0, 0) is saved to the random-access .DAT data file and any notes are saved to the .NOT notes file. SaveRec is limited to one note field per form.

  Usually, the OpenFiles routine is called before using either GetRec or SaveRec.

- Example

  Please *Saving Records* under *Performing Data Entry* for an example.

ScrollIn

BASIC subprogram contained in SCROLLIN.BAS

- Purpose:

  Scrollin is a pollable virtual field input routine that allows editing text that is wider than the window showing on the screen.

- Syntax:

  ```
  CALL ScrollIn(Edit$, Scroll)
  ```

- Where:

  Edit$: The string to be edited, and may range from 1 to 32000 characters in length

  Scroll: A TYPE variable that contains the remaining Scrollin parameters. These are assigned as follows:

  Scroll.Start — On entry, specifies which character in Edit$ to be placed at the left edge of the edit window.

  Scroll.Wide — The width of the edit window.

  Scroll.MaxLen — The maximum allowable length of the edited text; MaxLen must be at least as great as Scroll.Wide; if Scroll.MaxLen = Scroll.Wide then scrolling is disabled.

  Scroll.Filter — Determines the type of text to be accepted by ScrollIn, and may be set to any of the following values:

  0 All keys will be accepted

  1 Integer characters "1234567890- "

  2 All numeric characters "1234567890ED,.+-/\"

  3 User defined*

  4 Converts all letters to upper case

  5 Capitalizes the first letter of each word

  * Characters to be accepted are assigned to the Filter3 CONSTant in the SCROLLIN.BAS module level code.

Scroll.Ky: Returns the ASCII code of the last key pressed; if an extended key was pressed, Scroll.Ky returns a negative value corresponding to the key's extended code; if **Esc** is pressed, ScrollIn restores Edit$ to its original contents; if the left mouse button is clicked outside the edit window, ScrollIn responds as if **Enter** were pressed but returns a value of 1000

Scroll.EdClr: The color to use while editing; the foreground and background colors are combined into a single integer using the following formula:

```
TextClr = Foreground + Background
  * 256
```

Scroll.NormClr: The color to use when editing is complete; the foreground and background colors are combined using the formula shown above

Scroll.Action: Determines how Scrollin is to be invoked:

0-ScrollIn is not polled. When called, ScrollIn takes control of the program until a terminating key is pressed (i.e., **Enter**, **Esc**, **PgUp** and so forth)

1-ScrollIn is polled When called, control is passed alternately between ScrollIn and the calling program. This lets the calling program monitor editing as it occurs.

Scroll.Row: The screen row for the edit window

Scroll.Col: The left-most screen column of the edit window

Scroll.CurCol: Current screen column

Scroll.Insert: Sets and returns ScrollIn's insert state. -1 = Insert ON

Scroll.Changed: Returns -1 whenever Edit$ has been modified

Scroll.KeyChar: Returns the last key pressed as a 2 byte string

The Scroll TYPE variable is defined in the SCROLL.BI include file.

If the length of the text is greater than the size of the edit window, the text may be scrolled right or left by using the standard cursor keys, or with the mouse by holding the left mouse button down on the left-most or right-most character in the edit window. All of the standard editing keys are supported; in addition **Alt-C** clears the field and **Alt-R** restores the field to its original contents.

The Scroll.Filter argument specifies which set of characters are to be accepted based on three filter masks. The first two are defined in SCROLLIN.BAS, using CONST strings named Filter1$ and Filter2$. You indicate which to use by setting Scroll.Filter to 1 or 2. If Scroll.Filter is assigned to 3, then ScrollIn will use a filter mask that you define. Simply define Filter3$ as shown at the start of the SCROLLIN.BAS source file. In fact, any of the three filter masks can be customized to accept whatever characters you define. ScrollIn will accept only characters contained in the specified Filter?$ string.

If you do not require a mouse for your application, the block of mouse code in SCROLLIN.BAS can easily be removed. Simply search for "MMM" and remove code as the comments indicate.

For ScrollIn to work properly, you must also include the SCROLL.BI include file in whatever module calls ScrollIn.

SetUp

BASIC include file contained in SETUP.BAS

- Purpose

  SETUP.BAS is an include file that defines several arrays used by Graphics QuickScreen routines as COMMON SHARED. It also detects the current monitor type, initializes the mouse, and sets several of the global GPDat%() array elements to their default values. (See appendix A for more information on the GPDat%() array.)

- Syntax

```
'$INCLUDE: 'Setup.BAS'
```

- Comments

  SETUP.BAS includes the COMMON.BI include file which actually dimensions the GPDat%() and Choice$() arrays as COMMON SHARED. SETUP.BAS then calls MultMonitor to determine the current monitor type. The value returned by MultMonitor is assigned to GPDat(31). Next, SETUP.BAS sets default color values in the GPDat%() array for the pop-up list box and scroll bar used for multiple choice fields. If a mouse is detected, GPDat%(73) will be set to true (-1).

  SETUP.BAS should be included in the main module of your program before the first executable statement. Since SETUP automatically includes the COMMON.BI include file, you do not need to include COMMON.BI in your main module.

ShowCursor

assembler subroutine contained in GFORMS.LIB

- Purpose

  ShowCursor turns on the mouse cursor, making it visible. If the cursor is currently visible, ShowCursor does nothing, and leaves the mouse cursor visible.

- Syntax

  ```
  CALL ShowCursor
  ```

- Comments

  For more information see the comments that accompany the companion routine HideCursor.

- See Also

  HideCursor

Tokenize

BASIC subroutine contained in GQSCALC.BAS

- Purpose

  Tokenize replaces field names with a padded fixed-length (23-character) string containing the field number.

- Syntax

```
CALL Tokenize(Calc$, Fld())
```

- Where

  Calc$: Formula string

  Fld(): Field information TYPE array (see *FLDINFO.BI*)

- Comments

  This routine is used internally so that field formula strings can be properly read. By replacing field name strings with field numbers, the routine can properly access data items in a form.

UnPackBuffer

BASIC subroutine contained in EDITFORM.BAS

- Purpose

  UnPackBuffer copies and formats information contained in the form array Form$(0, 0) and fills the Form$(FldNo, 1) data array for each field.

- Syntax

```
CALL UnPackBuffer(FirstFld%, LastFld%, Form$(), Fld())
```

- Where

  FirstFld%: Starting field to be redisplayed

  LastFld%: Ending field to be redisplayed

  Form$(): Form string array (see *Form$() array*)

  Fld(): Field information TYPE array (see *FLDINFO.BI*)

- Comments

  This routine is useful when employing random access files to store and retrieve the contents of the Form Buffer, Form$(0, 0). If you are not using random access files then this routine is not needed.

  One important note is that UnPackBuffer places information into the Form$() array only, and does nothing with the screen. To update the screen with the contents of each field you must also CALL the PrintArray routine.

- Example

  This example fills the Form$(FldNo, 1) array element for fields five through ten with information contained in the form buffer:

```
CALL UnPackBuffer(5, 10, Form$(), Fld())
```

Value

BASIC function contained in EDITFORM.BAS

- Purpose

  Value returns the value of a numeric string.

- Syntax

```
StringValue# = Value#(NumString$, ErrorCode%)
```

- Where

  NumString$: Numeric string

  ErrorCode%: Returns 0 if no overflow occurred; -1 if an overflow occurred

- Comments

  Because Value has been defined as a function, it must be declared before it may be used.

  This function is used to convert a numeric string to a double-precision number. The numeric string can contain such characters as dollar signs, commas, exponent signs, and so on. If an overflow occurred when the string was being converted to a number, then the ErrorCode value will be -1; otherwise it will be 0.

- Example

  This example extracts the value 5000 from the string "$5,000":

```
StringValue# = Value#("$5,000",ErrorCode%)
```

WholeWordIn

BASIC subroutine contained in GQSCALC.BAS

- Purpose

  WholeWorldIn locates a substring within a string, using math operators as delimiters.

- Syntax

  ```
  CALL WholeWordIn(Text$, Word$)
  ```

- Where

  Text$: String to be searched

  Word$: Word to be search for in Text$

- Comments

  This routine is used internally so that field names and other "words" can be found in formula strings.

Routines

8


Development
in QB/QBX

DEVELOPING IN THE BASIC ENVIRONMENT

Whether you are using QuickBASIC or the QBX editor that comes with Microsoft BASIC Professional Development System, you'll need to make certain environment variables and library routines available to the compiler environment. The steps needed to prepare the environment properly are summarized below.

1. Switch to the Graphics QuickScreen directory so that BASIC source (.BAS) and include (.BI) files are in the current directory. If include files are elsewhere, be sure to set the INCLUDE environment variable properly.

2. If you want to create .EXE files from within the environment, you should place all .LIB files in the same directory. Then, make sure that the LIB environment variable is set to the correct directory path. If the linker cannot locate needed .LIB files, it usually generates an "Unresolved external reference" error.

Environment variables are usually set from a batch file or directly at the DOS prompt. The suggested method, however, is to add such commands to your system's AUTOEXEC.BAT file. This way, environment information will be established each time the computer is booted.

If you prefer, you can create a batch file in your Graphics QuickScreen directory which you can run before your Graphics QuickScreen sessions. An example SET command is:

```
SET LIB=C:\QB\LIBS
```

3. In order to run Graphics QuickScreen programs in the environment it is necessary to load a Quick Library which, at the very least, contains the assembly language routines that Graphics QuickScreen requires.

Only one Quick Library can be used, and it must be loaded when starting BASIC. Furthermore, loading Quick Libraries in either QB or QBX reduces the amount of conventional RAM available, so it is important to keep such libraries as small as possible. We suggest using the MakeQLB utility which we have included.

4. If you are using AJS Publishing's db/LIB® product, you can add it to a Quick Library by specifying its .LIB file when using MakeQLB.

5. Upon starting QB or QBX, you can load a Quick Library by using the /L command line switch. The following examples illustrate how a Quick Library called MYQLB is loaded for QuickBASIC or QuickBASIC Extended, respectively:

```
QB /L MYQLB
```

or

```
QBX /L MYQLB
```

6. Once QuickBASIC is started you should inspect the currently-set path options. To do this, access the (Options) Set Paths... menu command (not all versions of QuickBASIC support this feature). You should ensure that all shown paths are accurate.

Programs that use Push Buttons or Mouse Fields

When mouse fields or push buttons are activated from the keyboard, the KeyDown function is used to determine when the key has been released. This function is actually a small TSR (Terminate and Stay Resident) routine that is installed and deinstalled by calling the InstallKeyDown and DeInstallKeyDown subroutines. When you use **Make Demo...** to generate your BASIC source code, declare statements and calls for these subroutines are automatically added to the code as required. If you generate your source code from scratch, you will have to declare and call these routines yourself. The syntax is very simple:

```
DECLARE SUB InstallKeyDown ()
DECLARE SUB DeInstallKeyDown ()
.
.
.
CALL InstallKeyDown
.
.
.
CALL DeInstallKeyDown
END
```

InstallKeyDown should be called anytime before calling EditFormG. The DeInstallKeyDown subroutine should be called just before you end your program. KeyDown is called internally by EditFormG and therefore requires no coding on your part.

When you are developing in the QB or QBX environments and break out of your program, then re-start it, you will find that KeyDown no longer

works. This is because DeInstallKeyDown must be called before your program ends. If this happens, the only way to get KeyDown to work again is to exit and then re-start QB or QBX. If your program ends normally, DeInstallKeyDown will be called as required and the problem will not occur. See the documentation for KeyDown under the *Graphics QuickScreen Routines* section of this manual.

Displaying Screens From Your Program

Screens are displayed in your programs by calling the ShowForm subroutine. ShowForm automatically sets the screen mode, adjusts the color palette to what it had been when the screen was saved and sets the number of screen rows. Partial .PCX screens saved by Graphics QuickScreen can be positioned at any row and column by setting the Row and Column arguments.

Before calling one of these routines, the Fld() TYPE array must first be dimensioned and assigned. This is discussed in the section *Assigning Field Definitions*. The Fld() array is passed to the ShowForm routine to let it change the fields coordinates when partial .PCX screens are repositioned, and to tell it the number of screen rows to set.

The easiest way to display and edit your forms from a BASIC program is to select **(Compose Fields) Make Demo...** . This will create a BASIC source file that you can run from BASIC and will behave as if **Try Data Entry in Form** had been selected. It will also setup appropriate SELECT CASE statements to handle push buttons and scroll bars as well as generate temporary choices for multiple choice fields.

A .MAK file is also created that contains all the required support modules or stub files necessary to display and edit your forms. To run the demo, exit Graphics QuickScreen and start QB or QBX with the appropriate Quick Library (GForms.QLB or GForms7.QLB). Use the **Open** command from BASIC's **File** menu to select the demo. Once loaded, you may run the demo by pressing **Shift-F5**, assuming all of the files specified in the .MAK file are in the current directory. This code serves as an excellent starting point for creating your own source code.

.PCX screens that do contain any field definitions can also be displayed by ShowForm, but in this case, you simply dimension the Fld() TYPE array to 0 before the call. In the case of display-only screens, the Fld() array passes only two useful values to ShowForm. These are Fld(0).Value and Fld(0).Indexed which tell ShowForm the correct screen mode and number of text rows to set respectively. If these variables are set to 0, ShowForm will use the current setting of GPDat%(31) to determine what

screen mode to set. (The GPDat%(31) element holds a value indicating the current monitor type. See Appendix A for more information about GPDat%(31) and the GPDat%() array.)

To force a specific screen mode, refer to the table below.

Screen Mode	Fld() Settings
EGA 640x350	Fld(0).Value = 5
25 lines	Fld(0).Indexed = 14
43 lines	Fld(0).Indexed = 8
VGA 640x480	Fld(0).Value = 8
30 lines	Fld(0).Indexed = 16
60 lines	Fld(0).Indexed = 8

Table 23: ShowForm Response to Fld(0) Values

The calling syntax for ShowForm is as follows:

```
CALL ShowForm (FileName$, Fld(), Row, Column, VPage,
  ErrorCode)
```

Here, Filename$ is the name of the .PCX file to display without an extension.

Row is the screen row in pixels used to locate the top row of a partial .PCX image. Since data entry fields must be located using standard text rows and columns, the number used for the Row argument should be a value that positions the image such that the fields will still fall on standard text coordinates. The original row and column coordinates for the upper left corner of partial .PCX images are stored in the form definition file. Fld(0).Row holds the screen's upper row in pixels and Fld(0).LCol holds the screen's left column. These values can be used to position a partial image in its original position:

```
CALL ShowForm(FileName$, Fld(), Fld(0).Row, _
   Fld(0).LCol, VPage%, ErrorCode%)
```

Any other value can be assigned for the Row% and Col% parameters as long as the entire form will still display. Showform will automatically adjust an incorrect Row% value to the nearest pixel in order to maintain

proper alignment. This lets you enter just about any value for Row%, but it does not necessarily place the image at the exact row specified. To calculate the exact row, multiply the number of text rows to move by the height of a standard text character (contained in the GPDat%(71) element). Then add (or subtract) the result to the original row contained in Fld(0).Row.

```
NewRow% = Fld(0).Row + 5 * GPDat%(71)
CALL ShowForm(FileName$, NewRow%, Fld(0).LCol - 4,_
  VPage%, ErrorCode%)
```

This example places the image five text rows down and four columns to the left of its original position.

Column indicates the left column (1-80) position of the partial .PCX image.

The Row and Column parameters should be set to 0 when displaying full screen images.

VPage indicates onto which video page the screen is to be loaded. The default visible video page is 0. With this setting, you will see the image wipe down the screen as it is loaded. Setting VPage to 1 will load the screen into the second video page without displaying it. This lets you display the screen with one of several wipe types discussed in the following section. Since few VGA video adapters contain enough memory for more than one 640x480 video page, the VPage parameter is ignored by VGA screens.

ErrorCode returns one of three values to indicate the success or failure of the display:

0 - No errors, display was successful

1 - Trying to display on an incompatible monitor

2 - Error in loading the .PCX file

The following example indicates the minimum code required for showing a display-only screen.

```
DEFINT A-Z
DECLARE FUNCTION MultMonitor%
DECLARE SUB ShowForm (FormName$, Fld() AS ANY, Row,_
  Col, VPage, ErrorCode)

'$INCLUDE: 'FieldInf.Bi'
'$INCLUDE: 'SetUp.BAS'
REDIM Fld(0) AS FieldInfoG

ShowForm "MyScreen", Fld(), 0, 0, 0, ErrorCode
```

Displaying EGA Screens With A WipeType

Wipe types allow you to display your EGA screens in a number of interesting ways. They are produced by loading the image onto the non-active video page and then restoring portions to the visible video page. Wipe types are not available for high resolution VGA screens because there is not enough video memory available to allow two video pages, nor are wipe types available for partial .PCX screens.

To display an EGA screen with a wipe type, use a value of 1 for VPage when calling ShowForm. This will load the screen into the non-visible video page. You can then call the Wipes subroutine to display the image with one of the available wipe types:


```
VPage% = 1
CALL ShowForm(FileName$, Row%, Column%, VPage%,_
  ErrorCode%)
CALL Wipes(WipeType%)
```

When Graphics QuickScreen generates your source code, it automatically sets VPage% to 1 and uses BASIC's PCOPY command to instantly copy the screen to the active video page. If you wish to use a wipe type, you must replace the PCOPY statement in the main module with a call to the Wipes subroutine.

The various wipe types are summarized in Table 24.

WIPETYPE	EFFECT
1	Displays the screen instantly
2	Implodes the screen
3	Builds the screen from 8 pixel blocks
4	Displays from the upper left corner to the lower right corner
5	Builds the screen from a series of squares #1
6	Builds the screen from a series of squares #2
7	Builds the screen from a series of squares #3
8	Builds the screen from a series of squares #4
9	Builds the screen from a series of squares #5
10	Pushes the screen down from the top and up from the bottom simultaneously.
11	Pushes the screen from the top down and from the middle of the screen down simultaneously.
12	Pushes the screen down from the top and up from the bottom simultaneously in four sections.
13	Displays the screen using a horizontal blind effect #1
14	Displays the screen using a horizontal blind effect #2
15	Explodes the screen from the center.
16	Slides the screen up from the bottom
17	Slides the screen down from the top
18	Slides the screen from right to left
19	Slides the screen from left to right
20	Slides four quadrants onto the screen from the center
21	Slides every other pixel line from left to right and from right to left to merge into a complete image.

Table 24: EGA Wipetypes

Since you will probably use only a few of these wipes in a given application, we suggest that you copy and paste only the ones you require from the EGAWIPES.BAS module into your source code and then call them directly. (Wipes is a BASIC subroutine contained in EGAWIPES.BAS)

Displaying .GMP Files

Files created with the **Save Paste Buff...** option under the File menu can be displayed from your BASIC programs using a combination of the GetGMP subroutine and BASIC's graphic PUT statement. GetGMP loads the image from disk and places it into an integer array. If an error occurs

while loading the image from disk, Errcode% will be set to -1. Before calling the GetGMP subroutine, you must create an array to hold the image by redimensioning it to 0 elements:

```
REDIM Image%(0)
CALL GETGMP(FileName$, Image%(), ErrCode%)
PUT (X, Y), Image%, PSET
```

If your program uses a mouse, you will need to turn off the mouse cursor before displaying the image, and then turn it back on afterwards. This is accomplished by calling HideCursor and ShowCursor respectively:

```
REDIM Image%(0)
CALL GETGMP(FileName$, Image%(), ErrCode%)
CALL HideCursor
PUT (X, Y), Image%, PSET
CALL ShowCursor
```

Once the image is loaded into the array, it can be placed at any X/Y position using PUT as many times as you like. This is the technique used to place icon images on the Graphics QuickScreen Drawing Palette. The coordinates that you specify must place the entire image on the screen or BASIC will issue an "Illegal Function Call" error. PUT can also display an image using one of several display attributes: PSET, PRESET, AND, OR and XOR. Consult your BASIC manual for more information on effect of these attributes.

Storing .PCX, .FRM, and .GMP files in a .GSL library

When you distribute programs created with Graphics QuickScreen, you must also supply the various screen (.PCX, .GMP) and form definition files (.FRM) that they require. To avoid distributing many individual screen and form definition files, you can combine your .PCX, .FRM and .GMP files all into a single custom library. You then only have to distribute a single library file along with your final .EXE.

Creating a Custom .GSL library

The GQSLIB utility program is provided to let create your own custom library for storing the various files created by Graphics QuickScreen. The library can hold any type of file, but routines to retrieve the information from the library are provided only for .PCX, .FRM, and .GMP files.

To build the library file, you must first create a list file that identifies the files you want to place in the library. The format of the list file is very simple and can be created in any text editor that generates a pure ASCII text file. (The BASIC editor works nicely). To create the list file, enter

each individual file name on its own line and include the complete path if the file is not located in the current directory:

```
C:\GQS\MyScreen.PCX
C:\GQS\MyScreen.FRM
C:\GQS\ICONS\MailBox.GMP
C:\GQS\InputBox.PCX
C:\GQS\InputBox.FRM
C:\GQS\MsgBox.PCX
Title.PCX
```

Save this file with a .LST extension. The name of the library that GQSLIB creates will have the same name as your list file but with a .GSL (Graphics QuickScreen Library) extension. The files may be listed in any order, and you can have up to 500 different files in a single library.

To create the library, run the GQSLIB program and specify your list file as a command line argument.

```
GQSLIB [Path]MyLib.LST
```

This example will create a library named MYLIB.GSL. If you later decide to add or delete files from the library, simply edit your list file as required and run GQSLIB again.

Accessing data in a .GSL library

Several BASIC subroutines are provided to retrieve the individual file information from the .GSL library and are contained in the LIBFILE.BAS module. Each routine in LIBFILE.BAS has the same name as the equivalent routine used for accessing individual files but is pre-pended with the letters "Lib". For example, the ShowForm subroutine is used to display individual .PCX files. When the .PCX files are stored in a .GSL library however, you would instead use the LibShowForm subroutine. The calling syntax for LibShowForm is identical to that for the ShowForm routine except that LibShowForm has one additional argument to specify the name of the library file. Similarly, GetGMP is replaced by LibGetGMP, GetFldDefG is replaced by LibGetFldDefsG, and NumFieldsG is replaced by LibNumFieldsG. For further details, see the documentation for each routine in the *Routines* section of this manual.


Development
in QB/QBX

9

Performing Data Entry


Performing
Data Entry

PERFORMING DATA ENTRY

Forms may be processed from your BASIC programs using several routines. When you wish to use forms you will need to know how to access and initialize field data, how to use multiple-choice fields, and how to poll the EditFormG routine. We suggest studying and experimenting with the commented demonstration programs included on the distribution diskette.

General Concepts

When using Graphics QuickScreen, it is important to understand that the screen image and a form with which it may be associated are separate files and are therefore handled independently. In fact, data entry can still take place even if the appropriate screen is not displayed. EditFormG simply uses whatever colors it finds on the screen to use with the current form definition file. Once a screen is displayed, forms can automatically direct data entry activity.

Form data can be stored in a three different ways. The first is a standalone file having the same name as the screen with which it is associated, but with a .FRM extension. The second is a BASIC module that contains the field assignments for your form. The third method stores the .FRM file in a custom .GSL library file.

Data Entry

EditFormG is a BASIC subprogram which handles all aspects of data entry in a form. It is used in a manner similar to INKEY$. Thus, EditFormG continually polls for input while in a loop. While looping, a program can perform other operations before and after each call to EditFormG. In this manner you can achieve multi-tasking behavior.

EditFormG always furnishes current information to the calling program. Since you can read fields or even change them, you can tailor the operation of any field in a form.

General Procedures

The following section explains how to write code to set up and edit Graphics QuickScreen forms in your BASIC programs. Remember that this code can also be created automatically by selecting **(Compose Fields) Make Demo...** while still in the Graphics QuickScreen editor.

DemoAnyG.BAS

This program provides a good starting point for understanding how Graphics QuickScreen forms are used from your own programs. Although the source code is commented, you will find additional information here.

First, we suggest that you set all numeric variables to integers by default, and that you declare the required BASIC and assembler routines—especially functions. These steps are accomplished as follows:

```
DEFINT A-Z
'-------- Declarations
DECLARE FUNCTION MultMonitor% ()
DECLARE FUNCTION NumFieldsG% (FormName$)
DECLARE SUB EditFormG (Form$(), Fld() AS ANY, _
  Frm AS ANY, Action)
DECLARE SUB GetFldDef (FormName$, StartEl%, Fld() _
  AS ANY, Form$())
DECLARE SUB ShowForm (ScreenName$, Fld(), Row%,_
  Col%, VPage% ErrCode%)
```

Next, you should load the necessary Include files at the beginning of your programs. These files supply constant definitions and TYPE variables that are needed by your programs.

The Include files required for this example are FLDINFO.BI (which contains the FieldInfoG TYPE and associated constants), EDITFORM.BI (which contains the FormInfoG TYPE and associated constants), and SETUP.BAS. BASIC source code which Includes these files looks like:

```
'$INCLUDE: 'FLDINFO.BI'
'$INCLUDE: 'EDITFORM.BI'
'$INCLUDE: 'SETUP.BAS'
```


Since the DEMOANYG.BAS module uses forms, you will need to dimension the Frm variable to the FormInfoG TYPE. This makes information about the current form available to your program and to the supporting Graphics QuickScreen routines:

```
DIM Frm AS FormInfoG   'FormInfoG is a TYPE variable
```

Before allocating memory to the arrays which are used to control the form, it is necessary to determine the number of fields that are present. When using standalone .FRM files (as in this example) you will need to use the NumFieldsG function. This is demonstrated in the following excerpt:

```
NumFlds% = NumFieldsG(FormName$)
```

The integer value assigned to NumFlds% is used to dimension the Fld() and Form$() arrays.

Another array which can be dimensioned at this time is Choice$(). This array's first subscript provides the maximum number of choices you will need for any multiple-choice field, while the second subscript is one less than the total number of multiple-choice fields in the form. If your form has no multiple-choice fields, the Choice$() array must be dimensioned to zero elements.

```
REDIM Fld(NumFlds) AS FieldInfoG
REDIM Form$(NumFlds, 2)
REDIM Choice$(0, 0)
```

The next step is to load the form definition file. Once again the routine you'll use depends on how the form was stored. For standalone (.FRM) files, you will need to use the GetFldDefG routine:

```
CALL GetFldDefG(FormName$, Zero%, Fld(), Form$())
```

If you instead assign the form definitions from the BASIC subroutine use this:

```
CALL MyProg(Fld(), Form$, Start%)
```

Next you can display the screen image with ShowForm:

```
CALL ShowForm(FormName$, Fld(), Row%, Col%, _
  VPage%, ErrorCode%)
```


At this point, the form has been displayed and its field definitions loaded. In order to allow input, the form has to be activated. This is done by calling EditFormG with an Action of 1, which sets up internal pointers and displays initial field values from the Form$(N, 0) array elements.

After the first call to EditFormG, Action will be automatically set to 3, allowing polling to continue. It is up to you to examine Frm.KeyCode so that certain keypresses can be recognized. In this example, both **Esc** (which has a keycode value of 27) and **F2** (which has a keycode of -60) are used to terminate the form:

```
Action = 1
DO
  CALL EditFormG(Form$(), Fld(), Frm, Action)
LOOP UNTIL Frm.KeyCode = 27 OR Frm.KeyCode = -60
END
```

Detailed Procedures

The prior section covered the fundamentals of the DEMOANYG program. The following sections provide slightly more detailed discussions for using forms.

Setting Up A Form

Before a form can be processed, there are some suggested as well as required steps which must be taken. The optional steps involve such details as clearing the screen before generating the form, printing explicit instructions to the user, and setting the insert status or other features in the Frm TYPE variable. It is beyond the scope of this user's guide to cover all such optional aspects of programming. Instead, we'll provide the basics below, and encourage you to experiment on your own, using portions of the included demonstration programs as building blocks for your own programs.

The best way to process forms in QuickBASIC is to follow the steps outlined below and discussed next.

1. Specify the necessary Include files
2. Dimension the mandatory arrays
3. Load and display the form
4. Initialize the field and form elements

Specify Include Files

As discussed earlier, you must make certain TYPE declarations, constant assignments, and COMMON SHARED variables available to the calling program. In addition, if you have generated .BI Include files for your forms, you may want to specify them at the top of your program.

The BASIC statements for required include files are summarized below:

```
'$INCLUDE:  'FLDINFO.BI'
'$INCLUDE:  'EDITFORM.BI'
'$INCLUDE:  'SETUP.BAS'
'$INCLUDE:  'MYFORM.BI' 'this Includes the TYPE
                        '  structure for your form
```

The COMMON.BI Include File

The COMMON.BI include file contains two COMMON SHARED arrays that are required by the routines that you will use to display and edit your forms. This file is automatically included and initialized in the mandatory

SETUP.BAS include file placed at the beginning of your program. It is also included in the routines that we supply, but you may want to include it in some of your own modules as well.

If your program requires any additional COMMON SHARED variables, they should be added to this file and included in all of your modules that need to access them. QuickBASIC and BASIC PDS require that all COMMON SHARED variables be listed in the same order for each module that uses them. Placing COMMON SHARED variables in a single file and including it in your code insures that their order will always be the same.

The two arrays declared as COMMON SHARED in the COMMON.BI file are GPDat%() and Choice$(). The GPDat%() array holds information that affects how your forms operate. See Appendix A for more specific information on the contents of the GPDat%() array. The Choice$() array holds the choices that appear in multiple choice list boxes. See *Setting Up Multiple-Choice Fields* for more information on the Choice$() array.

Dimension Mandatory Arrays

The arrays which must be dimensioned are among those discussed in the section called *Arguments*. Although these arrays are dimensioned to zero elements at the start, you should realize that all are rediminsioned later when needed.

The required dimension statements are shown below, and they rely on the Include files mentioned in the previous section:

```
DIM Frm AS FormInfoG
REDIM Fld(0) AS FieldInfoG
REDIM Form$(0, 0)
REDIM Choice$(0, 0)
```


Load The Form

Form information is loaded into the form arrays using GetFldDefG or by calling the BASIC *MYFORM* subroutine optionally created when the form was saved. The GetFldDefG routine uses the Fld() and Form$() array variables and requires that each be dimensioned before the call. These arrays are dimensioned using information from the NumFieldsG function. If you have Graphics QuickScreen create the BASIC subroutine for you, the arrays are dimensioned automatically.

Once the form is loaded, its screen image is displayed using the ShowForm subroutine discussed earlier.

These program instructions are summarized below.

```
StartEl% = 0
Size% = NumFieldsG%(FormName$)
REDIM Fld(Size%) AS FieldInfoG
REDIM Form$(Size%, 2)
CALL GetFldDef(FormName$, StartEl%, Fld(), Form$())
'now display the screen
```

Initialize Field And Form Elements

Before and while a form is being used you may set certain field and form values. For example, you could use the Form$() array to assign default values to particular fields. Or you could use the Frm TYPE variable to set the form's insert status. These form variables were presented in the *Arguments* section of this manual. We have chosen to illustrate here how to optionally set the insert status of the form, how to set a multiple-choice array, and how to create a few default field values.

Setting The Insert Status

To set the insert status you will need to use the Frm TYPE variable. The following statement initially sets a form's insert status to off:

```
Frm.Insert = 0          'set insert off
```

Setting Up Multiple-Choice Fields

If you are using multiple-choice fields in your form, there are certain measures which you must take for them to work properly. First, you must include the module LISTBOX.BAS instead of NOMULTG.BAS. This way the full ListBox subprogram will be available to your program.

Multiple-choice fields require initializing the COMMON SHARED string array Choice$(). (Note that this array has already been defined as COMMON SHARED in the COMMON.BI file). Choice$() is a two-dimensional array which is dimensioned as follows:

```
REDIM Choice$(MaxChoices%, NumChoiceFields%)
```

MaxChoices% is the maximum number of choices which any list box would need to hold. The first subscript element (that is, Choice$(0, N)) is always reserved for the field number(s) to which the multiple-choice array is will linked.

The second subscript, NumChoiceFields%, tells how many unique multiple-choice menus are needed. NumChoiceFields% begins at element 0 (some fields share the same multiple-choice information, so there could be more multiple-choice fields than this subscript indicates).

For example, suppose a form has only 2 multiple-choice fields and that the first, field number 2, is for soft drink selections while the second, field number 6, is for T-Shirt sizes. Let's also suppose that there are 5 soft drinks and 3 T-shirt sizes available.

One way to redimension and initialize the Choice$() array for this example would be:

```
REDIM Choice$(0 to 5, 0 to 1)
Choice$(0, 0) = "2"          'Choices for field 2
Choice$(1, 0) = "Pepsi"
Choice$(2, 0) = "Coke"
Choice$(3, 0) = "Dr. Pepper"
Choice$(4, 0) = "7-Up"
Choice$(5, 0) = "Canada Dry"

'Choices for fields 6, 15 and 16
'  choice array
Choice$(0, 1) = "6, 15, 16"
Choice$(1, 1) = "Small"
Choice$(2, 1) = "Medium"
Choice$(3, 1) = "Large"
```

As you can see, Choice$() element (0, N) shows which field uses the list of choices in subscripts (1, N), (2, N), (3, N), and so forth. Further, this element can specify that several fields will share the same series of items. This keeps the Choice$() array as small as possible.

Setting List Box Colors

You can define a single set of colors for all list boxes on the form or allow each list box to use the colors assigned to its associated field. The setting of the COMMON SHARED variable GPDat(90) determines what set of colors are to be used for the text portion of the list box.

For a single set of list box colors, set GPDat%(90) to 0 before calling EditFormG. You can then specify the list box text colors by setting the following GPDat%() array elements:

GPDat%(91) = Normal text color + background color * 256

GPDat%(92) = Highlight text color + background color * 256

By default, GPDat%(91) and GPDat%(92) are assigned in the

SETUP.BAS include file to use black text on a gray background for list boxes. SETUP.BAS include file.

If GPDat%(90) is set to -1, the list box will appear in the same colors as its associated field. (These colors are inverted to create the highlight bar.)

The list box will display a scroll bar if the number of choices exceeds the value of GPDat(99). The default value is set to seven, but you may specify any number greater than six. You can change the colors used for the list box scroll bars by changing the settings of the following GPDat% array elements. The default color assignments in SETUP.BAS are for standard gray.

```
GPDat%(87) = Scroll bar push button color  (7)
GPDat%(88) = Scroll bar highlight color    (15)
GPDat%(89) = Scroll bar shadow color       (8)
GPDat%(98) = Scroll bar slide color        (7)
```

Creating Default Field Values

To assign default values into fields you will assign elements in the Form$() array (see *Form$() array*). You will generally modify only the field data in Form$(); however you may also change the help text and formulas as well.

For instance, suppose that field 1 holds the current date and field 4 holds a tax rate. The calling program could initialize these fields like this:

```
Form$(1, 0) = DATE$
Form$(4, 0) = "7.5%"
```


One last comment regarding Form$() is that for notes (or multi-line text) fields the entire field is returned as a single string. Blank lines embedded in notes fields are returned as a CHR$(20).

If you modify the contents of a field in the Form$() array after editing has begun, you must call either EditFormG with Action set to 1 or PrintArray with the appropriate field number to display the new contents. EditFormG redisplays the contents of the entire form while PrintArray can be used to redisplay only a specified range of fields.

Using EditFormG

Once a form is properly initialized, you will call EditFormG to perform data entry. EditFormG works much like BASIC's INKEY$, and, as such, it is designed to be called repeatedly in a loop. When a routine is called in this way it is said to be polled. Polling offers a tremendous level of flexibility since the calling program becomes an extension of the processing logic. For instance, polling lets you modify a form on-the-fly, that is, while the form is being used.

EditFormG processes each keystroke or mouse event immediately, and then returns to the calling program. Often, however, EditFormG receives no input whatsoever, and simply returns to the calling program for its next iteration. What's important is that control is returned to the calling program between keystrokes and mouse events, so you can monitor input as it occurs.

Earlier we discussed the form variables: the Form$() string array, the field information Fld() TYPE array, and the form information Frm TYPE variable. Each variable contributes or provides information about a form, and, together, these variables can be both accessed and changed while a form is being used.

These important form variables are reviewed below, in the context of how they can be used with EditFormG.

Form$()

In general, the Form$() string array (see *Form$() array*) is used to pre-fill a form or to examine data provided by the user. If needed, data in other fields can be changed based on information already entered. For example, if one field on your form is for Gender, and accepts "F" for Female and "M" for Male, then a subsequent field for a Salutation could be automatically pre-filled with "Mrs." or "Mr.", respectively. Doing this with EditFormG is easy, as you'll see later.

Fld() TYPE Array

The Fld() TYPE array (see *FLDINFO.BI*) determines and controls field attributes. Changing this array lets you protect fields or change valid data ranges for specific fields. For instance, an invoice form might ask for a salesperson number. After this field is complete, you could protect it so that once the order is taken, the salesperson's code cannot be changed.

Frm TYPE Variable

The Frm TYPE (see *EDITFORM.BI*) variable provides more general information about a form, such as whether any information on it has changed, which field is currently-active, and which key was last pressed. Often it is desirable to change where the cursor is on a form, based on certain data. For example, if an order form allows purchases with a credit card, it usually also has fields for the credit card number and its expiration date. However, if the sale is paid for by check, you may want to automatically skip over the credit card fields and jump to the Check Number field.

In the following program fragment, EditFormG is initially called with an Action of 1, which fills and initializes the form. Then, after each polling cycle, the program checks to see if the user has left the current field. This test is accomplished by comparing the value of Frm.PrevFld (the previous field number) with Frm.FldNo (the current field number). This test is true for one polling cycle only—after this, Frm.PrevFld becomes Frm.FldNo. If the test is in fact true, the program enters a SELECT CASE block which examines the value of Frm.PrevField to see which field the user just left.

In this example, the program checks whether "F" or "M" has been entered into the Gender field (field number 5). If the field is null, then no action is taken. If it contains an "F", then the Salutation field (field number 8) is set to "Mrs.". It is likewise set to "Mr." if the Gender field contains an "M":


```
Action% = 1
DO
  CALL EditFormG(Form$(), Fld(), Frm, Action%)
  IF Frm.PrevField  Frm.FldNo THEN
    SELECT CASE Frm.PrevField
      CASE 5           'did we just leave field 5?
        IF Frm.FldEdited
          IF Form$(5, 0) = "F" THEN
           Form$(8, 0) = "Mrs."
          ELSE
           Form$(8, 0) = "Mr."
          END IF
        END IF
        Action% = 1    'this forces the next call to
                       ' EditFormG to refresh the form
      CASE ELSE
      'CASE ELSE is needed with QB 4.0 only
    END SELECT
```

```
    END IF
  LOOP Until Frm.KeyCode = 27
```

There are many tricks you can perform by manipulating the Graphics QuickScreen form variables which have been presented. One common example is building shortcut keys into a form so that a user can skip large portions over several fields at a time. For complex forms this is useful. To illustrate, you could decide that field 10 is to be associated with the **F2** key, while field 20 is to be associated with the **F3** key. This way, you could examine Frm.KeyCode each time EditFormG is called. You could then test for **F2** or **F3** and set Frm.FldNo so it points to a new field. This would allow a user to move instantly between fields 10 and 20 with a single keystroke.

```
SELECT CASE Frm.KeyCode
  CASE -60                'They pressed F2
    Frm.FldNo = 10        'Go to field 10
  CASE -61                'They pressed F3
    Frm.FldNo = 20        'Go to field 20
END SELECT
```

Navigating A Form

Fields are accessed in a form through a number of methods. Fields may be selected by clicking on them with the mouse or by using the **TAB**, **Shift-TAB**, or cursor direction keys. The effect of each key is summarized below:

TAB	Moves forward through a form one field at a time.
Shift-TAB	Moves backwards through a from one field at a time
Right-Arrow	Moves forward to the next field when the cursor is at the end of the current field
Left-Arrow	Moves backwards to the previous field when the cursor is at the beginning of the current field.
Up-arrow	Moves to the field above the current cursor position
Down-Arrow	Moves to the field below the current cursor position

For the **Up** and **Down** arrows keys to work correctly, no field may be located higher than the first field on the form or lower than the last field on the form. The **Up** and **Down** arrow keys can be disabled by setting the UpDnArrows constant in the EditForm module to 0.

When you reach the end of a data entry field, the setting of the StayOnField Constant in the EditForm module determines whether the cursor stays in the current field or automatically jumps to the next field. When set to True (-1), the cursor remains in the current field. The default value is False (0).

Random-Access File I/O

The most direct way to save and load form information is in random access files. This method relies on the fixed-length structure of the form buffer, Form$(0, 0). Using random access files lets you create a single database file and quickly access any record it contains.

Graphics QuickScreen provides an example of random access files in the DEMOCUSG.BAS demonstration program.

Random Access File Setup

To use a random access file you will need to open the data (.DTA) file. This is achieved using the OpenFilesG routine, which also opens associated notes files and ensures that the form buffer is properly sized to accommodate random-file data. This step is summarized as follows:

```
CALL OpenFilesG(FormName$, Form$(), Fld())
```

Once the data files are open and the form arrays are initialized, you can use the GetRecG and SaveRecG routines to load and save records, respectively. Before retrieving records, you should know the upper limit—that is, the total number of records currently stored. This value is determined by dividing the current length of the data file by the record length of the form being used:

```
LastRecord& = LOF(Fld(0).RelHandle) \ Fld(0).StorLen
```

Notice that the OpenFilesG routine assigns a BASIC file number for the associated .DTA file to Fld(0).RelHandle. Although you probably won't need to access it, OpenFilesG also assigns to Fld(0).ScratchI the handle for the .NOT notes file.

Retrieving Records

To retrieve a record from the data file, simply call the GetRecG routine giving a valid record number in RecNo:

```
CALL GetRecG(RecNo&, Form$(), Fld())
```

This loads only Form$(0, 0) with information. You will need to use the UnPackBuffer routine to transfer form buffer contents to the individual form data elements:

```
CALL UnPackBuffer(FirstFld%, LastFld%, Form$(),_
   Fld())
```

To fill the entire form with information from the form buffer, FirstFld should be 1, and LastFld should be assigned to the total the number of fields in the form. The total number of fields is found in Fld(0).Fields. If you wish only to fill a portion of the form with information from the form buffer, you can specify other values for FirstFld and LastFld, realizing also that several ranges of fields can be filled by calling UnPackBuffer several times with different values.

One entirely optional step, which serves only to simplify programming and increase readability of your source code, is to copy information from the form buffer to a TYPE array corresponding to the current form. This lets you refer to fields by the name in the TYPE structure, rather than by accessing the Form$() elements by number. This is shown in the example that accompanies the BCopy routine description.

The next step is to copy the information in the form data elements onto the screen. The easiest way to do this is to call EditFormG with an Action of 1 (see *Action* for details). This also ensures that all of the fields that are displayed on the form are current. You can optionally call the PrintArray routine, which provides more flexibility by allowing a specific range of fields to be updated.

Saving Records

Record data is written to disk by calling the SaveRecG routine. All you need to do is furnish the record number in RecNo:

```
CALL SaveRecG(RecNo&, Form$(), Fld())
```

SaveRec writes information to the .DAT and .NOT data files opened earlier by the OpenFiles routine.

Clearing A Form

The best way to clear a form is to set each Form$(N, 0) element to a null string. You may then want to pre-assign certain fields in the form before allowing a new record to be entered. The code fragment below clears the Form$() data elements, sets the date and time information, and sets Action to 1 before EditFormG is called again.

```
FOR N = 1 TO Fld(0).Fields   'Clear all fields
  Form$(N, 0) = ""
NEXT N

Form$(12, 0) = DATE$         'Set date/time info.
Form$(15, 0) = TIME$
Action = 1                   'Prepare to refresh with
                             '  EditFormG
```

Notes Fields

Notes fields (sometimes referred to as MEMO fields) have a variable length and they are stored in a separate file. Although you could set aside a certain number of bytes for a notes field in each record database, that would be wasteful for records that have no notes or only a few characters.

Using Notes

If you plan to use the multi-line notes feature in your form, you will need to replace the NONOTESG.BAS stub file module with the GQEDITS.BAS module.

Data entered into notes fields is stored in a separate file with a .NOT extension. For each note field in a form, the Form$(0, 0) array element contains a long-integer pointer into the notes data file. This pointer accesses a two-byte integer in the notes file which gives the length of the string stored in the following bytes. The next note in the notes file immediately follows, and also begins with a two-byte integer giving the length of the note text, and so on. This arrangement keeps the notes file as compact as possible.

If you want to pre-fill a notes field with data, all you need to do is to fill the Form$() array with text. Recall that text is stored as a single line of information, using a CHR$(20) to indicate a blank line. Thus, to pre-fill field 6 in your form, you would write to Form$(6,0):

```
Form$(6,0) = "This is line one of a note field _
   "+ CHR$(20) + "and this is line two."
```

Saving And Retrieving Notes Data

The best way to both save and recall information from notes fields is to use the SaveRecG and GetRecG routines in the RANDOMG.BAS BASIC module.

Relational Fields

Graphics QuickScreen provides ways to store information which can be used to create relational database fields. These are fields which are common to two or more forms. For example, a customer number could be the record key to a customer file, and the same customer number may also appear as a field in an invoice file. Related fields make it easy to access information stored in different data files which is linked together by a common field.

Relational fields eliminate duplication (and thus wasted disk space) in your data files. As long as the customer number is stored in the Invoice file, there's no need to also store the customer's name and address there too. That information can be read from the Customer field when needed, using the Customer number field in the Invoice file to find the corresponding record in the Customer file.

While Graphics QuickScreen does not directly support related data files, it does offer a means by which your own programs can store and access information for related fields.

If you want to create a related field for field N in the current form, you would need to store information in the Fld(N).RelFile and Fld(N).RelFld field array elements. These two type elements describe the related file and field names, respectively.

If you want to access the related file by its file number rather than by its name, you can open the file and store its BASIC file number in the Fld(N).RelHandle TYPE array element. Using file numbers provides fast access, but this technique requires the file to be already open.

Indexed Fields

As with related fields, Graphics QuickScreen does not provide support for field indexing. However, it does provide a way to indicate that certain fields are indexed. You can use this to know if a given field is indexed when looking up records in the file. For instance, if field N of the current form is indexed, you would set Fld(N).Indexed to show that. Indexes are used to accelerate record searches and sorts, and most books on database design explain them in detail.

Note that if your form uses scroll bars, the scroll bar's small change value is stored in Fld(N).Indexed. You should therefore exclude scroll bars when searching the Fld() array for indexed fields:

```
FOR i = 1 to UBound(Fld)              'For each field
  If Fld(i).FType < ScrollBarFld then 'is it a
                                      ' scroll bar?
    'This must be an indexed field
  END IF
NEXT
```

(ScrollBarFld is a CONSTANT defined in EDITFORM.BI)

Multi-Page Forms

One important feature of Graphics QuickScreen is its ability to manage multi-page forms. This feature exploits the ability of the Frm() and Form$() arrays to hold several forms at once. Each "page" of a multi-page form is a standalone screen that is created separately in the Screen Designer.

Screens are designed so that they appear to visually follow one another—either from top to bottom (for tall forms) or from left to right (for wide forms). Using various wipe effects, you can achieve the illusion of moving up and down through a long form. Thus, pressing **PgUp** could scroll a new page down from the top of the screen using the "SlideD" wipe, while **PgDn** could make use of the "SlideU" wipe. Even though each screen is entirely separate, each accesses a unique range in the form arrays, making multi-page forms possible.

To illustrate the need for multi-page forms, suppose you want to allow 60 lines for item-entry on an invoice form. In 25-line mode, you would need three different "pages" to contain all of the data. The first page would contain header information, such as customer information, and would also begin the columnar section which forms the line-item section of the invoice (this is where part numbers, descriptions, and price information is entered). The second page would probably consist entirely of the line-item section of the invoice. The last page would complete the line-item section, and also have "footer" information, such as price totals and special shipping instructions.

The DEMOPAGG.BAS demonstration program shows the basics of multi-page form processing and of course includes commented source code. It serves as an example, and, as written, is limited to a two-page form stored at the beginning of a form library file. The discussion which follows attempts to give a more generic approach to handling multi-page forms which may exceed two pages.

Implementation

From a programming perspective, the concept behind multi-page forms is simple: place the names of your forms in a string array. This lets you refer to each screen by number and your program can increment and decrement a screen counter to access the next or previous screen image, respectively. When a user presses **PgDn**, or moves beyond the last field on the current "page", you will increment the screen counter and display the next "page" of the form. Likewise, pressing **PgUp**, or moving beyond the first prompt, will access the previous "page".

The next step is to fill the Fld() TYPE array and the Form$() string array with the field data stored in the .FRM or MYFORM.BAS file.

Now that the form arrays are loaded and properly initialized, you can call the EditFormG routine. While using EditFormG, if **PgUp** is pressed then Frm.StartEl is assigned to one less than the starting field number for the current form. If **PgDn** is pressed then Frm.StartEl is assigned to one greater than the highest field number on the current form. Thus, a calling program can check Frm.StartEl to determine whether either **PgUp** or **PgDn** has been struck.

Because a multi-page form creates an extra element in the form arrays for each form "page", we suggest using the Form$(0, 0) form buffer (rather than the Form$(N, 0) data elements) when writing or reading from the data file.

Alternatively, you can also check Frm.Keypress for specific keystrokes (such as **PgUp**, **PgDn**, or function keys you wish to use) to determine whether the user is trying to access a prior or next "page" in the form.

This first technique (that uses Frm.StartEl) is demonstrated below:

```
.
.
.
Action% = 1
DO                          'Poll the editing procedure
  CALL EditFormG(Form$(), Fld(), Frm, Action%)
  'If the user pressed PgUp or PgDn or moved off
  '  the top or bottom of the form, "StartEl" will be
  '  updated by "EditFormG" so we need to check it.
  '  The last value is saved in "LastStartEl" for
  '  use as a comparison.

  '--Did page change?
```

```
    IF Frm.StartEl <> LastStartEl% THEN
      '--Previous page?
      IF Frm.StartEl < LastStartEl% THEN
        'Yes, set previous page number
        Scr% = Scr% - 1
      '--Next page?
      ELSEIF Frm.StartEl LastStartEl% THEN
        'Yes set next page number
        Scr% = Scr% + 1
      END IF

      '--Display the screen
      CALL ShowForm(FormName$(Scr%), Fld(), Row,_
                    Col, VPage, ErrorCode%)
      '--Save the new "StarEl"
      LastStartEl% = Frm.StartEl
    END IF                  'Keep editing until the user
                            '  presses the Escape key.
  LOOP UNTIL Frm.KeyCode = 27
```

If you are using the optional BASIC field definition modules to assign field definitions, you will have to make minor source code modifications to them to create a multi-page form. The Fld() and Form$() arrays are dimensioned at the beginning of each field definition subroutine. (These arrays are named Fd() and F$() in the source file.) The module containing the field definitions for the first page of your form should be modified to dimension the Fld() and Form$() arrays to the total number of fields in your multi-page form. You then need to remove the REDIM statements from the remaining field definition modules. When calling these field definition modules, you must set the Start% parameter to the appropriate element within the entire Fld() array so that the field definitions are loaded into the correct elements.

Programming Tips

The following sections provide additional programming tips which you may find both interesting and useful.

Manually Manipulating Form Data at Runtime

Since EditFormG is polled, the calling program has the opportunity to change data and examine keystrokes during data entry. The example below shows how to update the time on the screen once each second. The time is printed on the first line near the right margin.

```
DO
  CALL EditFormG(Form$(), Fld(), Frm, Action)
  IF CLNG(TIMER)  T& THEN    'Display the time
                             '   each second to
    T& = TIMER               '   show how things
    LOCATE 1, 70, 0          '   can be done while
    PRINT TIME$;             '   a form is being
  END IF'   edited.
LOOP UNTIL Frm.KeyCode = 27  'Keep editing until
                             '   user presses Esc.
```

When this routine executes, the time is updated while the form is accepting user input. This example demonstrates how two activities (processing a form and updating the time) can appear to occur simultaneously.

Assigning Variables To Refer To Fields

The Graphics QuickScreen FldNum function converts field names to numbers and makes it unnecessary to change program code when your form is modified.

For instance, if Form$(9, 0) currently refers to a customer phone field, and you add two new fields before that, the customer phone field would become Form$(12, 0). If Form$() array subscripts are used to access a field's data, it would be necessary to change array subscripts throughout your program. In the example provided, all Form$(9, 0) references would need to be changed to Form$(12, 0) so that they accurately reflect the new location of the phone field.

Referring to fields using variable names is easy. Consider this program fragment:

```
DateFld% = FldNum%("INVDATE", FLD())
  .
  .
  .
Form$(DateFld%, 0) = DATE$
```

This example shows how the variable DateFld is assigned to the field number corresponding to the field called "INVDATE". If the form changes and the field position of INVDATE is altered, this method ensures that the correct field receives the DATE$ information.

Updating Form Data Using SaveField

To update the Form$(0, 0) data buffer, the SaveField routine should be called. Recall that this routine verifies data in the specified field before copying it to the form buffer.

```
CALL SaveField(DateFld%, Form$(), Fld(), BadFld%)
```

Recalculating Fields Using CalcField

If you change a field by changing a Form$(N, 0) element which affects a calculated value somewhere on the form, you will need to call CalcFields so that all of the fields are properly recalculated. CalcFields is described in the *Routines* section of this manual.

Converting Formatted Strings to Numbers

To quickly convert a formatted string to a double-precision number you can use the Value function. If you need to extract a number from the IEEE string imbedded in the form buffer, you can use the appropriate conversion scheme based on the following:

```
Num% = CVI(MID$(Form$(0, 0), Fld(FldNo).Fields, 2))
Num& = CVL(MID$(Form$(0, 0), Fld(FldNo).Fields, 4))
Num! = CVS(MID$(Form$(0, 0), Fld(FldNo).Fields, 4))
Num# = CVD(MID$(Form$(0, 0), Fld(FldNo).Fields, 8))
```

In each line above, MID$ is used to access a specific number of bytes within the form buffer. The starting character position, or offset, into Form$(0, 0) is supplied by Fld(FldNo).Fields. Then the appropriate number of bytes are read, such as two for integer values, four for single precision, and so forth. Once the string is read, the CVx operation converts the string to a number and assigns the result to Num.

Redisplaying Form Data Using PrintArray

After you have made the desired changes to the form data, you can redisplay information in the form by calling the PrintArray routine. PrintArray is described in the *Routines* section of this manual.

Handling Mouse Fields

Mouse fields are activated by clicking on them with the mouse, pressing **Enter,** or by pressing a key that returns the pre-assigned key code when the button is currently selected. The key code is returned as soon as the button is released when using the mouse or after pressing **Enter** and is returned instantly whenever the pre-assigned key is pressed.

The value is returned in Frm.KeyCode variable and can be used along with the Frm.FldNo variable to determine when a mouse field has been activated. The following example assumes that field number 10 is a mouse field and has been assigned to return a key code of -67 (**F9**):

```
IF Frm.Fldno = 10 AND Frm.KeyCode = -67 THEN
   'They clicked the push button assigned to F9
   '  or pressed F9
END IF
```

If all of the fields in a form are mouse fields, it will probably be unnecessary for you to test for the field number.

If you are using the mouse field as a toggle check box, the corresponding Form$(N, 0) array element will hold an X when it is checked or a space when it is not. To activate a mouse field from code, assign an X to the corresponding Form$() array element to select (highlight) the field or assign a space to deselect (un-highlight) it. You must then call EditFormG with Action set to 1 or call PrintArray with the appropriate field number to display the new status.

Handling Push Buttons

Push buttons are activated by clicking on them with the mouse, pressing **Enter**, or by pressing a key that returns the pre-assigned key code when the button is currently selected. The key code is returned as soon as the button is released when using the mouse or after pressing **Enter** and is returned instantly whenever the pre-assigned key is pressed. The value is returned in Frm.KeyCode variable and can be used along with the Frm.FieldNo variable to determine when a button has been pressed. The following example assumes that field number 5 is a push button, and that it has been assigned to return a key code of -68 (**F10**):

```
IF Frm.Fldno = 5 AND Frm.KeyCode = -68 THEN
  'They clicked the push button assigned to F10 or
  '  pressed F10
END IF
```

If all of the fields on the form are buttons, it will probably be unnecessary for you to test for the field number.

Handling Scroll Bars

Scroll bar values are returned in the Fld(N).Value variable. Simply assign this value to the appropriate variable in your program. The following examples assume that a scroll bar has been assigned to field number 25:

```
Tempo = Fld(25).Value      'Assign the "Tempo" value
```

In most cases you will only want to respond to the value if it has changed. In that case, assign a variable to remember the previous value, and compare it with the current value:

```
IF Fld(25).Value <> LastValue THEN 'has it changed?
  LastValue = Fld(25).Value   'yes, remember the value
  Tempo = Fld(25).Value       'assign the new tempo
END IF
```

Normally, a scroll bar returns its maximum value when the pointer is at the bottom of a vertical scroll bar or at the right side of a horizontal scroll bar. To make the value of a scroll bar read in the opposite direction, subtract Fld(Frm.FldNo).Value from Fld(Frm.FldNo).HiRange before assigning your variables:

```
IF Fld(Frm.FldNo).Value <> LastValue THEN
  LastValue = Fld(Frm.FldNo).Value
  Tempo = Fld(Frm.FldNo).HiRange -Fld _
    (Frm.FldNo).Value
END IF
```

You may also re-assign any or all of the scroll bar's settings at runtime. You can set new high and low limits, small and large change values or reposition the scroll pointer. After changing any of these variables you must either call PrintArray with the appropriate field number or reset Action to 1 before the next call to EditFormG. Either of these methods will reset the scroll bar to its new settings.

```
IF ChangeValues then          'Set new Scroll values
  Fld(25).HiRange = 1000      'Set new upper limit
  Fld(25).LoRange = -200      'Set new lower limit
  Fld(25).Value = 150         'New pointer position
  Fld(25).RelFld = 50         'New large change value

  Fld(25).Indexed = 1          'New small change value
CALL PrintArray(25, 25, Form$(), Fld()) 'Reset
                                        ' scroll bar
END IF
```


Note that calling PrintArray redisplays only the fields specified. Setting Action to 1 will accomplish the same thing but redisplays all fields on the form and is therefore somewhat slower.

You may also reset a scroll bar's value by assigning the desired value to the Form$(N, 0) array, where N = the scroll bar's field number. In this case, you must call EditFormG with Action = 1.

The color used when clicking on the scrolling portion of a scroll is assigned according to the setting of GPDat%(100). The default is to use whatever color was used for the shaded portion of the scroll bar's push buttons. You can also disable a highlight color or specify any other color. See appendix A, *The GPDat%() Array* for more information on setting GPDat%(100).

Changing The Color Of The Mouse Cursor

The standard mouse cursor appears as a white arrow with a black outline. That is, color 15 (white) and color 0 (black). To change the color all you need to do is change the palette settings for either of these colors. If you still need to use white or black in your programs, just reassign any of the other colors to white or black. The **Palette Editor** makes this very simple.


Performing
Data Entry

10


Standalone
Programs

CREATING STANDALONE PROGRAMS

Standalone .EXE programs are created by linking your compiled BASIC program (object files) using the version of LINK supplied with BASIC. Compiling programs for standalone use is relatively easy. However, you must first be aware of BASIC Make files before using the compiler and linker.

MAKE Files

Make files with a .MAK extension are created by the BASIC environment whenever a program requiring more than one module has been saved. They are ASCII files with a .MAK file extension, and they simply list the names of other modules which must be present in order for the main module to run. All these modules must be compiled to object files and then linked together with the GFORMS.LIB library or GFORMS7.LIB when using BASIC 7 PDS.

Compiling Modules

BASIC source files are compiled using the BC.EXE command line compiler like this:

```
BC MYPROG.BAS /O/S;
```

This will create an object file named MYPROG.OBJ, assuming the program compiled successfully. You will then need to compile each BASIC module listed in your program's .MAK file in turn.

Linking

Once you have compiled all of your programs, you need to create a final standalone .EXE program. This is done by linking object files with the provided GFORMS.LIB library or GFORMS7.LIB when using BASIC 7 PDS.

If you are compiling and linking manually from DOS, then you will specify all your BASIC-compiled object modules, along with GFORMS.LIB (or GFORMS7.LIB), like this:

```
LINK PROG1.OBJ+PROG2.OBJ,,NUL,GFORMS[7].LIB
```

If you prefer you can start LINK without any options, and wait for it to prompt you for the information it needs.

You may also specify more than one library when linking. For example, if you need assembler routines from both GFORMS.LIB and our QuickPak Professional, you would tell LINK to use both of them:

```
LINK PROG1.OBJ+PROG2.OBJ,,NUL,GFORMS[7] PRO[7]
```

You may also add single object modules when linking, even if they are not present in a library at all:

```
LINK PROG1.OBJ+PROG2.OBJ+MYOBJECT.OBJ,,NUL,_
  GFORMS[7] MYSTUFF
```

If you prefer to combine several libraries into a single .LIB file, that is quite easy too. Although the LIB library manager is usually employed to add or remove object modules, you may also add one or more complete libraries like this:

```
LIB LIBRARY1.LIB+LIBRARY2.LIB+LIBRARY3.LIB
```

One useful link option you should be aware of is the /E command line switch. When LINK is invoked with /E, it creates an .EXE file in a special packed format. Not unlike the various archive programs, the code and data are compressed to take up less disk space. When the program is run, the first code that actually executes is an unpacking routine that puts everything back together again. The /E switch is specified like this:

```
LINK /E PROG1.OBJ+PROG2.OBJ,,NUL,GFORMS[7]
```

A packed program will require less disk space, however it of course requires the same amount of memory when it is run.

11

Utilities

GRAPHICS QUICKSCREEN UTILITIES

Screen Capture Program

PCXCAP.EXE is a TSR (Terminate and Stay Resident) utility that was written using Crescent's P.D.Q. product. PCXCAP occupies very little memory and allows you to capture any graphic screen* from within other application programs.

In order for PCXCAP to run properly, you must start it from DOS before running the program whose screens you wish to capture. Before running PCXCAP, you should consider the following points which apply to all TSR programs:

1. It should not be installed from a program that has shelled to DOS

2. When using more than one TSR program, the last TSR installed must be uninstalled first

Using PCXCAP consists of a few simple steps which are summarized below.

1. Run PCXCAP from DOS

2. Start another program from which screens are to be captured

3. When the desired screen appears, press **Alt-S**

4. Specify a name for the screen, and press **Enter**

Screen names are limited to eight characters with no extension. Therefore you can save them to the current directory only, and the program will append the .PCX extension for you. Since the screens are saved in the .PCX format, you can easily load them into the Graphics QuickScreen editor. (This assumes of course that the screens were displayed and saved in a Graphics QuickScreen compatible screen mode, i.e. 640x350 or 640x480 16 color.)

One note of warning: PCXCAP uses the screen name you specify and does not caution you if you will overwrite an existing screen with the same name. For this reason, be extremely careful when naming PCXCAP screens to be saved.

When PCXCAP is no longer required, you may remove it from memory by exiting all active applications to return to the DOS prompt. Then, run PCXCAP again using the "/U" command-line switch at DOS like this:

```
PCXCAP /U
```

* PCXCap will not work while running under Microsoft Windows.

Converting From QuickScreen To Graphics QuickScreen

The QS2GQS program converts existing QuickScreen text mode screens and their corresponding .FRM files into equivalent graphics mode screens. The graphic screen is saved as a .PCX file and the .FRM file is modified to work with the Graphics QuickScreen editor. Once a screen has been converted it may be loaded into the Graphics QuickScreen editor for further enhancements.

When you run QS2GQS.EXE, it displays a dialog box that prompts you for the following information:

- QuickScreen .SCR or .QSL Files:

  Enter the name of the screen (.SCR) file or the screen library (.QSL) file name. Be sure to include a complete path name if the file is not in the current directory.

- Form Name (for .QSL files only):

  No entry is required if you are converting .SCR files. For .QSL library files, enter the name of the screen you wish to convert. A path name should not be given.

- Convert to (.PCX, .FRM):

  Enter the drive, directory, and file name for the converted screen. Since a modified .FRM file is created, make sure you save the file to a different name or directory to prevent overwriting the original.

Once you have entered the required information click the OK command button. The utility will read the screen and its form definition files into memory. A second dialog box will appear allowing you to select the desired graphics mode. The dialog box's option buttons will default to the closest matching screen mode based on the number of screen rows required. You can of course select any screen mode you prefer.

If the your screen has more rows than the selected screen mode allows, the screen will be clipped as necessary at the bottom. Field definitions that would be located beyond the last row are placed on top of each other and will have to be relocated manually in the Graphics QuickScreen editor. If the screen has fewer rows than the new selected screen mode, the converted screen is positioned at the top of the screen leaving the bottom portion blank.

Click the OK button and your screen will be quickly converted and displayed in the graphics mode you selected. Once the conversion is complete, you may continue with additional conversions or press **Esc** to exit the utility.

Note that this utility cannot read QuickScreen .QFL files. You must supply individual .FRM files for each form that you convert.

Quick Library Make Utility

To simplify the creation of custom Quick Libraries, we've included a utility called MAKEQLB.EXE. This utility examines a program and all its dependent modules, and creates a new Quick Library containing only those routines that are necessary. This is important when the programs you develop are very large, because it eliminates the wasted memory taken by routines that are not used. MAKEQLB also lets you easily combine routines from multiple library files, without having to extract each individual object module.

MAKEQLB knows which routines are to be included by examining your main program for CALL statements, and by searching for DECLARE statements when the CALL keyword is not used. MAKEQLB also searches include files (even when nested, where one file includes another) and the .MAK file if one is present, to account for all of the modules in a complete program.

MAKEQLB also reports any subprograms or functions that have been declared but are not being used. Of course, those routines will not be added to the resultant Quick Library. It will also report all subprograms and functions that are present but never called. As an option, you may specify a file that contains a list of all the routines that are to be included in the library, rather than having MAKEQLB examine your source files.

MAKEQLB uses an interface similar to the LINK and LIB programs, and you may either enter the parameters on a single line, or wait for MAKEQLB to prompt you for them. The command line syntax is as follows:

```
MAKEQLB mainprog, qlbname, listfile, lib1 lib2,_
   bqlbname
```

You may also specify more than one file name to be examined, by separating each with a blank space:

```
MAKEQLB mainprog1 mainprog2, qlbname, listfile,_
   lib1 lib2, bqlbname
```

Mainprog is the main BASIC program to examine, with a .BAS extension assumed. If a file name with a .LST extension is given, MakeQLB will instead use the procedure names contained in that file when creating the Quick Library.

The qlbname parameter is the name of the resultant Quick Library. If the name is omitted, the library will have the same name as the main program, but with a .QLB extension. If indeed you omit qlbname, be certain to retain the delimiting comma. If you specify NUL for the qlbname, MAKEQLB searches for unnecessary DECLARE statements and dead code, but will not create a Quick Library.

The list file that is created contains a list of all the routines that are being added to the Quick Library. This file defaults to a .LST extension, and is in the correct format that MAKEQLB requires to create a library from a list of procedure names. This way, if you need to add a routine or two to the Quick Library later on, you can simply edit the generated .LST file. Creating a Quick Library from a list file is of course much faster than examining an entire BASIC program. If the listfile parameter is omitted, the same name as the main program will be used, but with an .LST extension. To tell MAKEQLB not to create a list file, use the reserved name NUL for that parameter.

The lib1 and lib2 parameters are .LIB library files that contain the procedures being added to the Quick Library. One or more library names may be specified, with a blank space used to delimit each name. If no library name is given, the name PRO.LIB is assumed.

The last parameter tells MAKEQLB which "bqlb" support library is to be specified when linking. The default name is BQLB45.LIB, which is the library that comes with QuickBASIC version 4.5. For other versions of BASIC, please see Table 25.

MAKEQLB works by creating an object file that contains the list of procedure names. By establishing these procedures as External, they will be included in the Quick Library automatically when MakeQLB invokes

LINK. The dirty work of extracting each routine from the various .LIB files is thus handled entirely by LINK.

BASIC version	BQLB .LIB File Name
4.0	BQLB40.LIB
4.0b	BQLB41.LIB
4.5	BQLB45.LIB
6.0	*depends on QB version number*
7.x	QBXQLB.LIB

Table 25: BASIC BQLB .LIB File Names

Utilities

12

Product Compatibility

Product Compatibility

COMPATIBILITY WITH THE GRAPHICS WORKSHOP, GRAPHPAK PROFESSIONAL, AND db/LIB

Routines from Crescent's Graphics Workshop or Graphpak Professional can be easily incorporated into your Graphics QuickScreen programs with a few simple modifications to the source code.

Graphics Workshop

Use the same standard code as found in the Graphics Workshop manual (pg 1-12). GETVIDEO.BAS is used in place of the SETUP.BAS include file. You will need to increase the size of the GPDat%() array from 86 to 100 elements. The GPDat%() array is dimensioned in GETVIDEO.BAS and should be modified as follows:

```
REDIM SHARED Tile$(0), AltTile$(0), GPDat%(100)
```

The SETUP.BAS include file also sets default colors for the help messages, list boxes, and list box scroll bar colors as follows:

GPDat(76) = 0 + 7 * 256	Listbox text color
GPDat(78) = 15 + 0 * 256	Listbox highlight color
GPDat(87) = 7	Listbox scroll bar colors
GPDat(88) = 15	Highlight
GPDat(89) = 8	Shaded portion
GPDat(90) = -1	Use field colors for multiple choice fields
GPDat(94) = 0	Message box text color
GPDat(95) = 7	Message box background color
GPDat(96) = 15	Message box highlight color
GPDat(97) = 8	Message box shade color
GPDat(98) = 7	Sliding portion of scroll bar

GPDat(99) = 7	Number of list items before displaying a scroll bar

These values can be added to your GETVIDEO.BAS include file by copying and pasting them from the SETUP.BAS file, or you may assign them directly in your source code any time after the '$INCLUDE: 'GETVIDEO.BAS' metacommand. Of course, you will only need to assign those elements that your program requires; if you are not using any multiple choice fields, you do not need to assign any Listbox or scroll bar colors. Message box colors are used when displaying help messages.

Replace the COMMON.BI include file in all modules with the COMMON.GW file from the Graphics Workshop. This can be accomplished by loading all of the required modules into the QB editor and using its global search and replace capability. You will also need to modify COMMON.GW by adding the Choice$() array to the list of COMMON SHARED variables:

```
COMMON SHARED GPDat%(), Font$(), FontWidth%(), _
   FontHeight%(), Choice$()
```

Calling ShowForm performs the same function as calling SetVideo, though you may still call SetVideo first.

GraphPak Professional

Replace the SETUP.BAS include file with SIMPLE.BAS. You will need to increase the size of the GPDat%() array from 63 to 100 elements. The GPDat%() array is dimensioned in GETVIDEO.BAS and should be modified as follows:

```
REDIM SHARED Tile$(0), AltTile$(0), GPDat%(100)
```

The SETUP.BAS include file also sets default colors for the help messages, List boxes and List box scroll bar colors as follows:

GPDat(76) = 0 + 7 * 256	Listbox text color
GPDat(78) = 15 + 0 * 256	Listbox highlight color
GPDat(87) = 7	Listbox scroll bar colors
GPDat(88) = 15	Highlight
GPDat(89) = 8	Shaded portion

GPDat(90) = -1	Use field colors for multiple choice fields
GPDat(94) = 0	Message box text color
GPDat(95) = 7	Message box background color
GPDat(96) = 15	Message box highlight color
GPDat(97) = 8	Message box shade color
GPDat(98) = 7	Sliding portion of scroll bar
GPDat(99) = 7	Number of list items before displaying a scroll bar

These values can be added to your GETVIDEO.BAS include file by copying and pasting them from the SETUP.BAS file, or you may assign them directly in your source code any time after the '$INCLUDE: 'GETVIDEO.BAS' metacommand. Of course, you will only need to assign those elements that your program requires; if you are not using any multiple choice fields, you do not need to assign any ListBox or scroll bar colors. Message box colors are used when displaying help messages.

Replace the COMMON.BI include file in all modules with the COMMON.BAS file from Graphpak Professional. This can be accomplished by loading all of the required modules into the QB editor and using its global search and replace capability. You will also need to modify COMMON.BAS by adding the Choice$() array to the list of COMMON SHARED variables:

```
COMMON SHARED GPDat%(), Font$(), FontWidth%(), _
   FontHeight%(), Choice$()
```

Calling ShowForm performs the same function as calling SetVideo, though you can still call SetVideo first.

db/LIB

Graphics QuickScreen provides four BASIC subroutines for interfacing Graphics QuickScreen with AJS Publishing's db/LIB. These routines are:

db2FormG	Transfers and converts data from a db/Lib record to the Form$() array for editing
dbDefineRecG	Defines a db/LIB record structure from a Graphics QuickScreen form definition

dbNumericStrG	Converts a formatted numeric string to a form compatible with db/LIB "N" field type
Form2dbG	Transfers and converts data from the Form$() array to a db/LIB record

These subroutines are contained in DBLIB_G.BAS. See the DEMODBLG.BAS demonstration program for an example of how these routines can be used in your program.

If you develop your program in the BASIC environment, you will need to make a combined Quick Library that contains the required library routines from both the GForms[7] library and the appropriate library from AJS. This can be accomplished by using the MAKEQLB utility supplied with this package. See the *QUICK LIBRARY MAKE UTILITY* form more details.

13

Trouble Shooting

TROUBLE SHOOTING

- The Mouse Cursor disappears

Occasionally, the mouse cursor may disappear after the selection of a new screen mode or after loading a new screen. If this should occur, the cursor can be brought back by pressing **Ctrl-F1**.

Computer hangs up when **Try Data Entry in Form** is selected or when EditFormG is called from your program.

This will happen if all of the fields in a form are protected. You must have at least one non-protected field on your form.

- You receive an "Out of stack space" error when EditFormG is called from your program.

This too will happen if all of the fields in a form are protected. You must have at least one non-protected field on your form.

- You receive an "Out of string space" error when you run your program from the BASIC environment.

You can increase the amount of memory available to your program by creating a custom Quick library that contains only the routines that your program requires. A smaller Quick library can be easily created using the MAKEQLB utility that is described elsewhere in this manual.

If you still receive the error, you can uses the various "No" stub files as you develop your program. Then when you compile and link your program, make sure to use the full-featured versions of the subroutines.

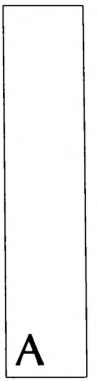
- You receive a "Subprogram Not defined" error.

This indicates that the routine being called has not been loaded. If the message refers to a BASIC routine, you must add the module that contains it to your program by using the **(File) Load** command. If the message refers to an assembler routine, then either the routine is not in the loaded quick library, or you failed to start BASIC with the required quick library. You can easily determine whether a routine is BASIC or assembler by looking up the routine in the Routines section of this manual. The heading at the top of the routine description indicates that the routine is either a "BASIC subroutine contained in *BASFILE*.BAS" or an "assembler subroutine contained in GFORMS.LIB"

- You receive the message "Cannot find the drawing palette's ICON.GMP Files" when activating the drawing palette or a "File not found" error when trying to access the tile palette or the scalable fonts.

This indicates that the various support files required by Graphics QuickScreen cannot be found. These files are installed initially in your Graphics QuickScreen directory. If you work from any other directory, change the path settings in the **Set Paths** dialog box found under the **Settings** menu to point to your Graphics QuickScreen directory (or the directory where they currently reside). These files are listed below:

Drawing Palette files:	SCRIBBLE.GMP, PBRUSH.GMP, BUCKET.GMP, ZOOM.GMP, AND CLRWHEEL.GMP
Tile Palette files:	TPAL.TIL, and TILEPAL.GM4
Font Files:	FUTURE.GFN, HELV12.GFN, HELV8.GFN, OLDENG.GFN, AND TROM12.GFN


A

APPENDIX A

The GPDat%() Array

The GPDat%() array is a COMMON SHARED array that contains information used by EditFormG to help process your forms. Users of Crescent's Graphic Workshop and GraphPak professional libraries are probably already be familiar with the GPDat%() array. Graphics QuickScreen uses the GPDat%(31) and GPDat%(71) elements as assigned by either of these libraries but expands the array with the addition of elements 87 through 100.

GPDat%(31)	The current monitor type; A value of 5 indicates EGA color; a value of 8 indicates VGA color; this variable is assigned in the SETUP.BAS include file
GPDat%(71)	ROM text height; this variable is set by ShowForm to the height in pixels of the current ROM font; possible values are 8, 14 and 16
GPDat%(72)	Video address for saving graphic images
GPDat%(73)	Boolean variable indicating the presence of a mouse; a value of -1 indicates that a mouse was detected
GPDat%(87)	Scroll bar push button color; This variable sets the scroll bar push button color when one appears on a list box; the default value is 7 and is set in the SETUP.BAS include file
GPDat%(88)	Scroll bar highlight color; this variable sets the scroll bar highlight color when one appears on a list box; The default value is 15 and is set in the SETUP.BAS include file
GPDat%(89)	Scroll bar shadow color; This variable sets the scroll bar shadow color when one appears on a list box; the default value is 8 and is set in the SETUP.BAS include file

GPDat%(90) Use field colors/GPDat colors for list box; This is a boolean variable that determines which set of colors to use when displaying a list box; when set to -1 the, list box appears in colors defined by GPDat%(91) and GPDat%(92); when set to 0, colors assigned to the multiple choice field are used for the list box; the default value is 0

GPDat%(91) List Box text color; the default value is black foreground on a gray background; the two colors are combined into a single integer using the following formula:

```
GPDat%(91) = fgcolor + bgcolor _
  * 256
```

Note that GPDat%(90) must be set to -1 for these colors to take effect.

GPDat%(92) List box highlight color; the default value is white text on a black background and is set in the SETUP.BAS include file; the two colors are combined into a single integer using the following formula:

```
GPDat%(92) = fgcolor + bground _
  color * 256
```

Note that GPDat%(90) must be set to -1 for these colors to take effect.

GPDat%(93) Save to video/conventional memory; this is a Boolean variable that determines where background screen images are saved when displaying list boxes and help messages; If set to -1, background screens are saved to conventional memory as an integer array; a value of 0 saves to video memory; the advantage of saving to video memory is that this memory is usually unused and does impinge on memory required by your program; the default value is 0; the only time to use a setting of -1 is when you are already using this memory for other purposes

GPDat%(94)	Message box text color; the default value is 0 (black) and is assigned in the SETUP.BAS include file
GPDat%(95)	Message box background color; the default value is 7 (gray) and is assigned in the SETUP.BAS include file
GPDat%(96)	Message box highlight color; the default value is 15 (white) and is set in the SETUP.BAS include file
GPDat%(97)	Message box shadow color. The default value is 8 (dark gray) and is set in the SETUP.BAS include file
GPDat%(98)	Scroll bar slide color; The default value is 7 (gray) and is set in the SETUP.BAS include file (for multiple-choice fields only)
GPDat%(99)	Number of menu items before displaying a scroll bar; sets the maximum number of items to display in a list box before displaying a scroll bar
GPDat%(100)	The color to use when clicking on the sliding portion of a scroll bar; values between 1 and 15 establish the color; when set to 0 the shade color of the scroll bar's push buttons is used; other values disable highlighting

GLOSSARY

GLOSSARY

calculated field

Any field in a form for which a formula has been defined.

cursor keys

Keys which control the movement of the cursor. These keys typically include the Up, Down, Left, and Right keys, and sometimes include PgUp and PgDn.

dialog box

An input screen which collects information needed for carrying out a process. For example, Graphics QuickScreen's **(File) Open...** pulldown command uses a dialog box.

dithered colors

A method of creating more than the standard 16 colors by alternating colored pixels within a block.

form

A screen which has at least one field defined.

.FRM files

A file containing field definitions for a form.

hotkeys

Keys which directly access an item on a menu bar or pulldown menu. The characters corresponding to hotkeys are usually underlined or highlighted.

insert mode

The edit mode in which each character to the right of the cursor is moved to the right as new characters are entered.

menu bar

A component of the menu system which presents pulldown menu names on the top line of the screen.

menu bar option

One of the menu names on the menu bar. A menu bar option usually presents a pulldown menu.

paste buffer

An area of memory to which graphic images are temporarily stored and retrieved.

pollable routine

A routine which may be repeatedly called in a loop. Pollable routines allow a calling program to carry out other tasks between each polling cycle, effectively simulating multi-tasking.

pulldown menu

A pop-up list of commands available for a given menu bar option.

ROM font

Your computer's internal font.

TUTORIAL

TUTORIAL

In the following tutorial, *clicking on* refers to pointing to an item with the mouse cursor and then clicking and releasing the left mouse button.

When you first start Graphics QuickScreen you are presented with a blank screen and a standard white arrow cursor. To design a screen with Graphics QuickScreen you will need to access the various paint and drawing tools available on the Drawing Palette. Try clicking the right mouse button several times to toggle the Drawing Palette on and off. The **Esc** key will also toggle the Drawing Palette.

Selections are made from the Drawing Palette by clicking on the desired color or tool icon. The selected color will appear in the lower right corner of the Drawing Palette. Selecting a tool clears the Drawing Palette and displays a drawing/editing cursor. Objects (lines, circles, boxes and so forth) are created by clicking on the desired starting position and then moving the drawing cursor to size the object. If you make a mistake, click the right mouse button once to cancel and begin again, or click twice to return to the Drawing Palette. Finish the object by clicking on the desired end point.

You may continue drawing with the same tool or you can click the right mouse button to return to the Drawing Palette. If you make a mistake, pressing **F10** will undo any drawing or editing done since the selection of the current tool.

Drawing and painting colors may be selected or changed at any time by pressing the color's corresponding numeric key. To access values above 9, hold down the **Shift** key to add 10 to whatever numeric key you press.

All of Graphics QuickScreen's features can be controlled from the keyboard or with a mouse in any combination. The cursor is controlled from the keyboard using the standard cursor keys. For drawing and editing procedures, pressing **Enter** is the same as clicking the left mouse button while pressing the **Esc** key is the same as clicking the right mouse button. In general, a left mouse click initiates or completes an action while a right mouse click cancels one. After any action has been canceled, the Drawing Palette can be accessed with one additional right mouse click.

When you select a drawing or editing tool, a drawing cursor will appear. As you move the cursor around the screen, you will notice that the drawing cursor does not move smoothly but rather skips from point to point on an invisible grid. This is called *grid snap*. On start up, grid snap is ON and

is set to spacings that correspond to the height and width of the text font for the current screen mode. Grid snap spacings can be set to almost any value but are particularly useful when used with the default settings. The reason for this requires some explanation.

When you design screens with data entry fields, a field's text is displayed using the computer's internal ROM font. Text is printed at standard text row and column coordinates using a solid background color. These fonts are always 8 pixels wide and are either 8, 14, or 16 pixels high depending on the screen mode. The text foreground color is defined when you assign field definitions, but the background color is read from the screen at run time. (In truth, the background color is set according to the color of the pixel located at the lower right corner of the first character position in the field.)

You will often want to paint a field's background using a color that contrasts with the surrounding screen to identify the field's boundaries. When you select the Filled Box icon with the default grid snap settings, any box you draw will be the exact size required to contain standard text.

Select a color and then click on the Filled Box icon (black rectangle) to select the Filled Box procedure. Try drawing several filled rectangles. When you are finished you can click the right mouse button to return to the Drawing Palette. Now select a new color and click on the "T" icon. A blinking text cursor will appear at its last used location. You can position the cursor with either the direction keys or the mouse and then begin typing. Notice that any text you type will line up with the blocks you just painted.

Grid Snap can be toggled on and off during any drawing or editing procedure by pressing the **S** key (grid **S**nap). Notice that the **Up**, **Down**, **Left** and **Right** cursor arrow keys always move the cursor in increments that correspond to the current grid snap settings. Cursor movement is therefore much faster from the keyboard when grid snap is turned on.

When the Drawing Palette is off you can click the left mouse button or press **Alt** to activate the menu system. When selected with the mouse, the last menu used will be displayed. Selections are made by clicking on them with the mouse, pressing the underlined or highlighted hotkey, or using the **Up** and **Down** arrow keys to highlight the desired choice and then pressing **Enter**. When activated by the **Alt** key, only the menu bar is activated. Menus can be accessed by pressing their underlined hotkey, pointing and clicking with the mouse, or by using the cursor keys and pressing **Enter**.

When a menu choice followed by an ellipses is selected, a dialog box will appear to allow you to enter additional information. Most of these dialog boxes can also be accessed directly without invoking the menu by pressing a corresponding function key.

Try experimenting with the various drawing and editing tools to get a feel for how Graphics QuickScreen works before moving to the next section.

Creating Data Entry Fields

For this exercise we will create three fields: a text field, a push button and a scroll bar. Make sure that grid snap is on and that the settings correspond to the current ROM font size. You can check the current settings by pressing **F8** to display the Status Box. The Status Box displays the current drawing color, snap status, and the current cursor position. Grid snap is on whenever the X/Y or R/C labels are displayed in upper case.

If grid snap settings correspond to the current ROM font size (referred to as text snap), a black rectangle (■) will appear in the Status box next to the Y or C label. If text snap is off, press the **F** key to turn it on. This key toggles between the current grid snap settings and appropriate text snap settings. You will hear two different beep tones as you toggle the **F** key. The higher pitch indicates that text snap is in effect.

If you want to clear the screen before starting this exercise, select **(File) New Screen...** . A dialog box will appear allowing you to select a new screen mode. Click OK after making your selection. The screen will be cleared to the current background color as assigned in the **System** dialog box under the **Settings** menu.

Activate the Drawing Palette and select the push button icon. Draw a single push button anywhere you like. If you wish to change colors or draw a larger push button, select a new color and draw another push button over the old one. You can also press **F10** to clear the image or use the Recolor option to change the push button's colors.

At this point, the push button is just a graphic image and will not function as a button until it is defined as a field. This will be discussed shortly.

Now select **Scroll Bar** from the **Draw** menu. Draw a scroll bar just as you drew the push button. If the defining rectangle is wider than it is high, a horizontal scroll bar is drawn. Otherwise, a vertical scroll bar is drawn. As with the push button, the scroll bar is just a graphic image until it is defined as a field.

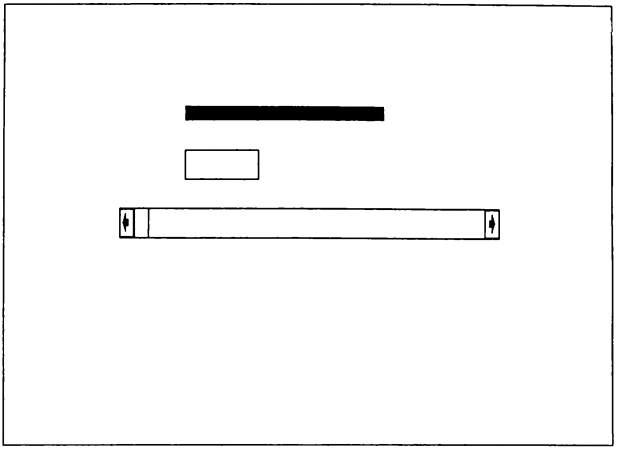


Figure 28: Example Screen

Next you will paint the background for a text entry field. Select the Filled Box icon (black rectangle) from the Drawing Palette and then draw a filled box one row high and any number of columns wide. The color you select will be the field's background color. Your drawing should now look something like figure 28.

The next step is to define these images as fields. Select **(Compose Fields) Enter Field Definitions...** . A message box will appear asking you to place the cursor at the beginning of the first field. Move the cursor to the beginning of the text field and double-click the left mouse button or press **Enter**. Another dialog box will appear allowing you to select the field type.

For now, select the default String type field by clicking on the OK button or by pressing **Enter**. A third dialog box appears asking you to locate the end of the field. Use the cursor keys or the mouse to adjust the size of the field. When you are satisfied with the size, double-click the left mouse button on the last character in the field or press **Enter**. A final dialog box lets you enter specific field settings. Accept the default settings by clicking the OK button or by pressing **Enter**.

When defining mouse fields, push buttons or scroll bars, the initial starting position of the field is irrelevant since it is redefined after the next step. Simply press **Enter** or double-click the left mouse button when prompted "Place the cursor at the beginning of the field". Select "Push Button" and click the OK button or press **Enter**. A message box then asks you to "Draw a box around the Push Button". As soon as you press a cursor key or move the mouse, the message will disappear.

Draw a box that *exactly* matches the black outline of the push button. With grid snap on this should be very simple. When the box is completed, the field settings dialog appears. For now, accept the defaults by clicking the OK button or by pressing **Enter**. Define the scroll bar using the same process but this time select "Scroll Bar" from the Field Type dialog box.

Once fields have been defined the form can be tested by selecting **(Compose Fields) Try Data Entry in Form**. Now the push button will depress when clicked on, the scroll bar will scroll, and you will be able to enter text in the text entry field. You can move from field to field by pressing **Tab** to move forward or **Shift-Tab** to move backwards through the form. You can also select any field with the mouse by clicking on it. When you are finished testing, press **Esc** to restore the screen and return control to the screen designer.

One very powerful feature of Graphics QuickScreen is its ability to copy a field or range of fields. Select the **(Compose Fields) Copy Fields** option. A message will appear asking you to define the fields to be copied by drawing a box around them. The box you draw may be any size, but only those fields whose coordinates fall completely inside the box will be copied.

For this exercise, copy the push button. Once you identify and capture the image, you can make as many copies as you wish. This procedure copies not only the graphic image of the button but also its field settings. Unique field names are generated automatically for each new field. Once you have completed making your copies, you can test the new push buttons by selecting **(Compose Fields) Try Data Entry in Form**. As you can see, this is an extremely fast way to generate duplicated field types.

Once you are satisfied with your form, Graphics QuickScreen can optionally generate a BASIC source file that you can run from the BASIC editor. This file will behave as if **Try Data Entry in Form** had been selected. Having Graphics QuickScreen create this portion of a program is a tremendous time saver because it automatically sets up the correct declare statements and include files for you. A .MAK file is also created that specifies all of the modules required to display and edit your form.

To create a BASIC demo, select **(Compose Fields) Make Demo...** . A dialog box will appear asking you to name the demonstration file, and whether the field definitions are to be loaded from disk (.FRM file) or hard-coded into your source code (.BAS module). The file name may be anything except the form name. Click OK to create the demo.

To run the demo, exit Graphics QuickScreen and start your version of BASIC with the appropriate library—GFORMS.QLB or GFORMS7.QLB. Use the **(File) Open...** command from the BASIC editor to load the demo. Once loaded, press **Shift-F5** to run it. Pressing the **Esc** key will end the program and return you to BASIC.

At this point, all of the form editing code has been written. You will still need to add code to handle values returned by mouse fields, push buttons, and scroll bars and to save or load the form contents.

This tutorial is provided to give you a quick understanding of some of the fundamentals necessary to use Graphics QuickScreen effectively. Only by reading the rest of this manual and examining and experimenting with the demonstration programs will you fully appreciate what Graphics QuickScreen can do.